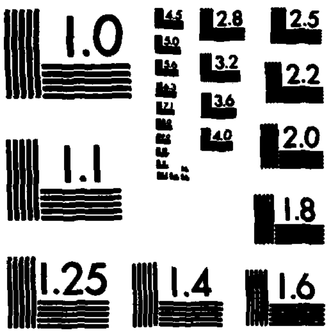
AD-A120 502 LSI APPLICATIONS(U) TRW DEFENSE AND SPACE SYSTEMS GROUP 1/1
REDONDO BEACH CA JUL 82 TRW-35735-6013-UT-00
RADC-TR-82-201 F30602-80-C-0150

UNCLASSIFIED F/G 9/2 NL

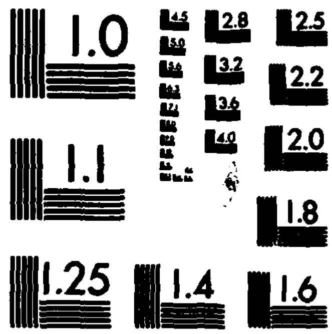
END

FILMED

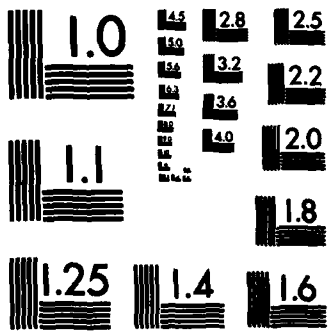
DTIC



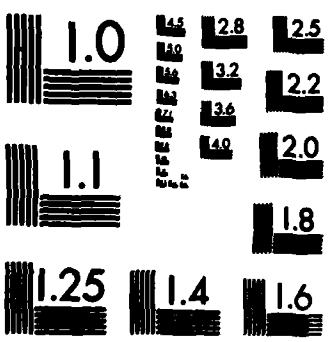
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



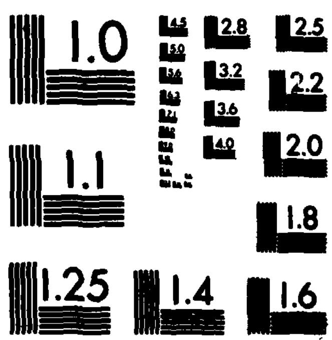
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

AD A120502

RADC-TR-82-201
Final Technical Report
July 1982

LSI APPLICATIONS

TRW

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER RADC-TR-82-201	2. GOVT ACCESSION NO. AD-A120502	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LSI APPLICATIONS		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report June 80 - June 82
		6. PERFORMING ORG. REPORT NUMBER 35735-6013-UT-00
7. AUTHOR(s) TRW		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0150
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW/Defense and Space Systems Group One Space Park Redondo Beach CA 90278		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63701B 32020324
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEA) Griffiss AFB NY 13441		12. REPORT DATE July 1982
		13. NUMBER OF PAGES 92
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Stephen M. Warzala (COEA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Emulation
Processor-Bound Computation
Computer Architecture
Microprogrammed Architecture
LSI Device Design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report deals with the task of achieving an order of magnitude increase in execution speed of a set of computation-bound processes. This increase was achieved by augmenting the host processor with a peripheral microprogrammed device. Both of these machines were emulated in a hardware description language. The peripheral device is composed of three major units:

(See Reverse)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

a. Microprogrammed Control Unit (MCU),

b. Arithmetic Processing Unit (APU), and

c. Interface Control Unit (ICU).

The structure and function of these individual units is examined in greater detail.

CONTENTS

CONTENTS (Concluded)

ILLUSTRATIONS

TABLES

1. INTRODUCTION

This document represents the final report of the LSI Applications project. As such, it will describe:

1. the analyses used throughout the project,
2. the results of the project, and
3. the conclusions and recommendations based on the previous analyses and results.

1.1 STATEMENT OF THE PROBLEM

The fundamental problem presented was to achieve, as a minimum, an order of magnitude improvement in performance in the map transformation algorithms used by the Defense Mapping Agency (DMA). Implicit in this overall requirement is a number of factors that will be discussed in the following sections.

1.1.1 Processor-bound Computations

The nature of the algorithms under consideration in this study is such that the performance bottleneck lies nearly entirely in the processing of the numerical algorithms themselves. The input/output (I/O), while certainly a large percentage of the total cost in the throughput equation, is an approximately fixed cost considering the type of storage media presently available. Although trends in storage technology will eventually permit a solution involving I/O techniques and media, present technology dictates that the currently-used I/O techniques and media be maintained. Thus the problem becomes one entirely associated with the processing cost of performing the algorithms. In this regard, a further consideration is that the major expense in this processing is the execution of numerical operations, rather than logical or control-type operations. Thus the problem further reduces to improving the numerical-calculation performance.

1.1.2 Computations on Different Machines

A further requirement levied is that the order of magnitude improvement in performance be obtained not only on one computing machine, but in two different types of machines. For this application the two machines considered were the PDP 11/60 and the Eclipse C-300. This requirement impacts the solution in that any proposed solution must take explicit account of the nature

of the two different computers in considering the costs involved in implementation. Exactly how this requirement impacts the proposed solution will be brought out throughout this report as each possible avenue is examined.

1.1.3 Order of Magnitude Improvement in Performance

Throughout this report, the goal of an order of magnitude improvement in performance is taken to mean a tenfold increase in throughput in data reduction by the DMA's transformation algorithms. Thus, any combination of improvement in algorithm design, improvement in algorithm execution (performance), and improvement in data transfers associated with the execution of the algorithms is a valid means to satisfy the goal. In fact, from a system point of view, the problem may be thought of as an attempt to gain a tenfold increase in throughput, rather than simply as a tenfold decrease in the execution time of the algorithms.

1.2 APPROACHES TO THE PROBLEM (level of augmentation of the native machine)

In any attempt to raise the throughput of a process, the various levels of improvement of that process include:

1. Altering the process itself to make it more efficient (faster),
2. Handing off portions of the process to some auxiliary processor, and
3. Handing off all of the process to some auxiliary processor.

These three options map directly into three levels of hardware augmentation of the host computers:

1. No augmentation,
2. Partial augmentation, and
3. Complete augmentation.

1.2.1 No Augmentation

In terms of the present project, achieving the goals with no physical augmentation of the host computers could conceivably be accomplished by either improving the algorithms themselves or using the inherent features of the host machines to better advantage than current design makes use of them.

1.2.1.1 Algorithm Improvement

While there is always the possibility that some improvement in mathematical structure or technique would allow the same results to be obtained by some more efficient means, this possibility is not likely within the time frame of this project. In fact, while a more efficient mathematical technique or algorithmic structure may be not only a possibility but a preferred technique eventually, the possibility of an improved technique or new formalism should be considered as a study to be conducted at a later time. As such, the algorithms as delivered to TRW are used without any alteration in either the logical structure or numerical technique.

1.2.1.2 Performance Improvement Through Microcoded Functional Subsets

Since the host computers contain user-microcode capabilities, a possible solution could include the execution of either portions of the algorithms or the entire algorithms by microcoded instructions. This solution assumes that the code currently used in the host computers at the DMA facility is ten times less efficient than could be developed. While it would be possible to achieve some improvement at the microcode level, this approach does not approximate the goal of the project since the ISA of these machines are well suited to processing problems typified by the DMA transformations.

1.2.2 Partial or Limited Augmentation

As part of the cost and performance tradeoff, the possibility that just part of the process could be offloaded to dedicated hardware must be examined. The degree of offloading is bounded from above by the cost of the proposed hardware and bounded from below by the expected level of improvement in performance that could be expected by offloading just a part of the algorithms into dedicated hardware.

In order to improve throughput, the portions of the algorithms that could be offloaded into dedicated hardware must be compared with the total effect on the system of such a configuration. That is, replacing a particular function with a faster, dedicated piece of hardware might not result in a total increase in throughput if accessing this special piece of hardware caused an increase in some particular phenomenon, such as bus access or processor interrupt. Consequently, select but self-contained portions of the algorithms

would have to be identified and offloaded into special hardware in an attempt to minimize the occurrences of these phenomena.

1.2.3 Total or Complete Augmentation

Just as partial augmentation might be defined to result in less additional hardware loading the host bus, total augmentation should not necessarily cause a perceptible increase in hardware load on the system. Referring to the case stated in the previous section, offloading an entire process, that is, total augmentation, could reduce system load by decreasing processor involvement in the process.

Any solution to the total system problem must take into account the possibility of completely offloading execution of the target algorithm into an auxiliary processor. The relevant questions that then arise become those that deal with the relative effectiveness of the auxiliary hardware compared with its complexity and the relative system cost of using the auxiliary hardware.

Finally, any proposed solution should contain self-consistent methods of imbedding the proposed auxiliary processor into the existing operating environment, without requiring significant modifications of the operating system or applications software.

2. DESIGN ANALYSIS

The application of hardware elements as replacements for software processes follows a certain set of design principles. For the most cost-effective solution, certain assumptions and system-level decisions are made early in the design process. These include, in the most general terms, the desired performance margin (how much reserve is available for burst activity and for future expansion), the desired processor margin (how much more complicated a task can the processor handle), the flexibility of interaction with the host machine, and the flexibility of adapting the device for attachment to other machines. Although these particular points are functionally peripheral to the design goals, any proposed design should not exclude them. Given two equally effective and economical designs, the more restrictive design would naturally be the less desirable.

2.1 ANALYSIS

The first step in formulating a hardware replacement of a software process is the careful evaluation of the features of the software process that determine its performance. In addition, the context in which the particular software process is contained must be examined not only to ensure that the proposed hardware solution contains the same environmental assumptions as did the software process, but also to determine if there are any particular features of the operating environment that the hardware unit may exploit in a way that is unavailable to the software process.

In addition, the relative effects of the various aspects of the process must be examined. This allows the determination of the degree of performance improvement available by trading off various aspects of the process.

2.1.1 Analyze Computational Process

Two features of any software process are processor-bound activity and I/O-bound activity. In analyzing these two aspects of total system throughput, some considerations include the amount of data, the type or format of data, and the type of processing that must be performed.

2.1.1.1 Data Transfer

The data in this study takes the form of a pair of floating-point values. Each floating point value consists of two 16-bit words in the host computers. Consequently, each expected transaction (where transaction is defined as

the execution of a transformation on a single set of points) involves the movement of a total of eight 16-bit words -- two input pairs and two output pairs. Thus, for an expected rate of 10^6 transactions per second, the system would have to provide an 8 Megaword/second transfer rate.

2.1.1.1.1 Required Response. The expected working environment, while not a batch environment, is nevertheless not a real-time environment. The required response is a secondary issue. The required response becomes a response sufficiently fast enough to avoid overly-long lags in transformed output.

2.1.1.1.2 Required Throughput. The throughput requirements are phrased relative to the throughput obtained by the software process alone on the host machines. These values are listed in Table 2.1. The goal of an order of magnitude improvement in performance translates into the values also listed in Table 2.1. The baseline performance figures are the result of executing the Fortran object code on the emulation of the PDP11/60 on the Nanodata QM-1.

2.1.1.2 Processing

Processing can be of two types: arithmetic functions, and control functions. The arithmetic functions break down rather naturally into two distinct types: simple (e.g., multiplication, division, addition, and subtraction) and "non-simple" (e.g., trigonometric, square root, and logarithms).

2.1.1.2.1 Arithmetic Functions. To best analyze the processing flows of the transformation algorithms, the various algorithms can be broken down into a series of graphs. Each node in one of these graphs, called data-flow graphs, corresponds to an arithmetic operation, while each edge represents the movement of data into or out of the particular node.

For example, the data-flow graph for the equation:

$$A = B + C / (B + E) + SQRT (B + E)$$

is shown in Figure 2.1.

Figures 2.2 and 2.3 are examples of data-flow graphs for some representative transformation algorithms.

Table 2.1. Algorithm Timing

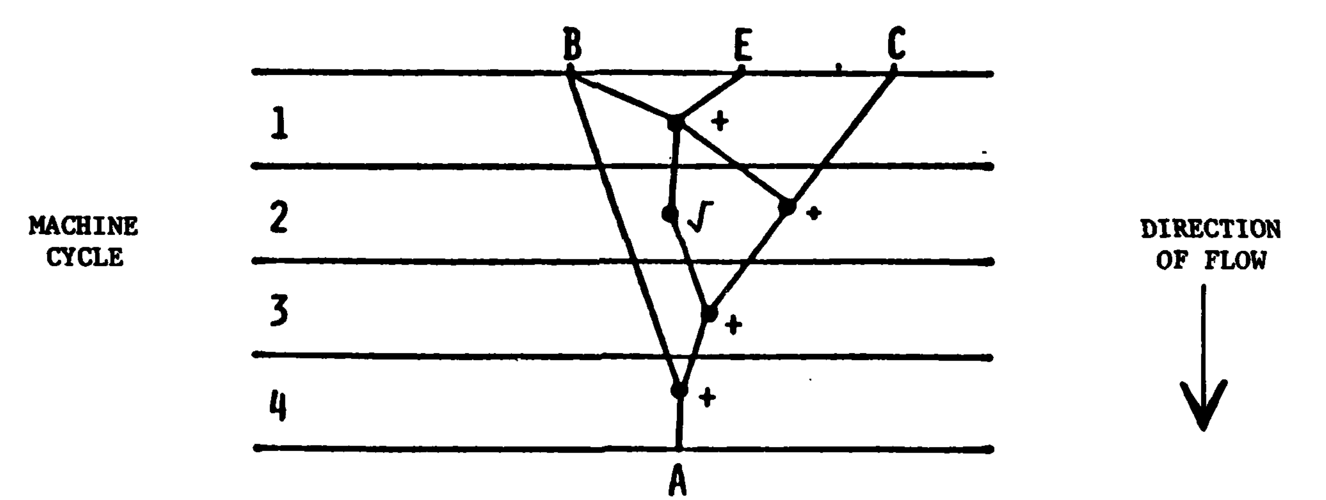
transformation routine	the execution speed of the transformation routine on the host	performance goal on the proposed device
mercator	1060 microseconds	106 microseconds
mercator inverse	700	70
transverse mercator	2190	219
transverse mercator inverse	4470	447
polyconic	1770	177
polyconic inverse	9560	956
lambert	1910	191
lambert inverse	1160	116

Note: The execution speed of the transformation routines are taken from measurements of the execution time of object code of the Fortran transformation routines running within an emulation of the PDP 11/60 on the Nanodata QM-1.

(1) Iterative. Assumes equal number of loops through the algorithm.

A = B + C / (B + E) + SQRT (B + E)

(B,C, AND E ARE INPUT PARAMETERS)



Assumptions:
1) only one functional unit of a kind
2) square root and divide can be performed in one machine cycle

Figure 2.1 Data-Flow Graph Example

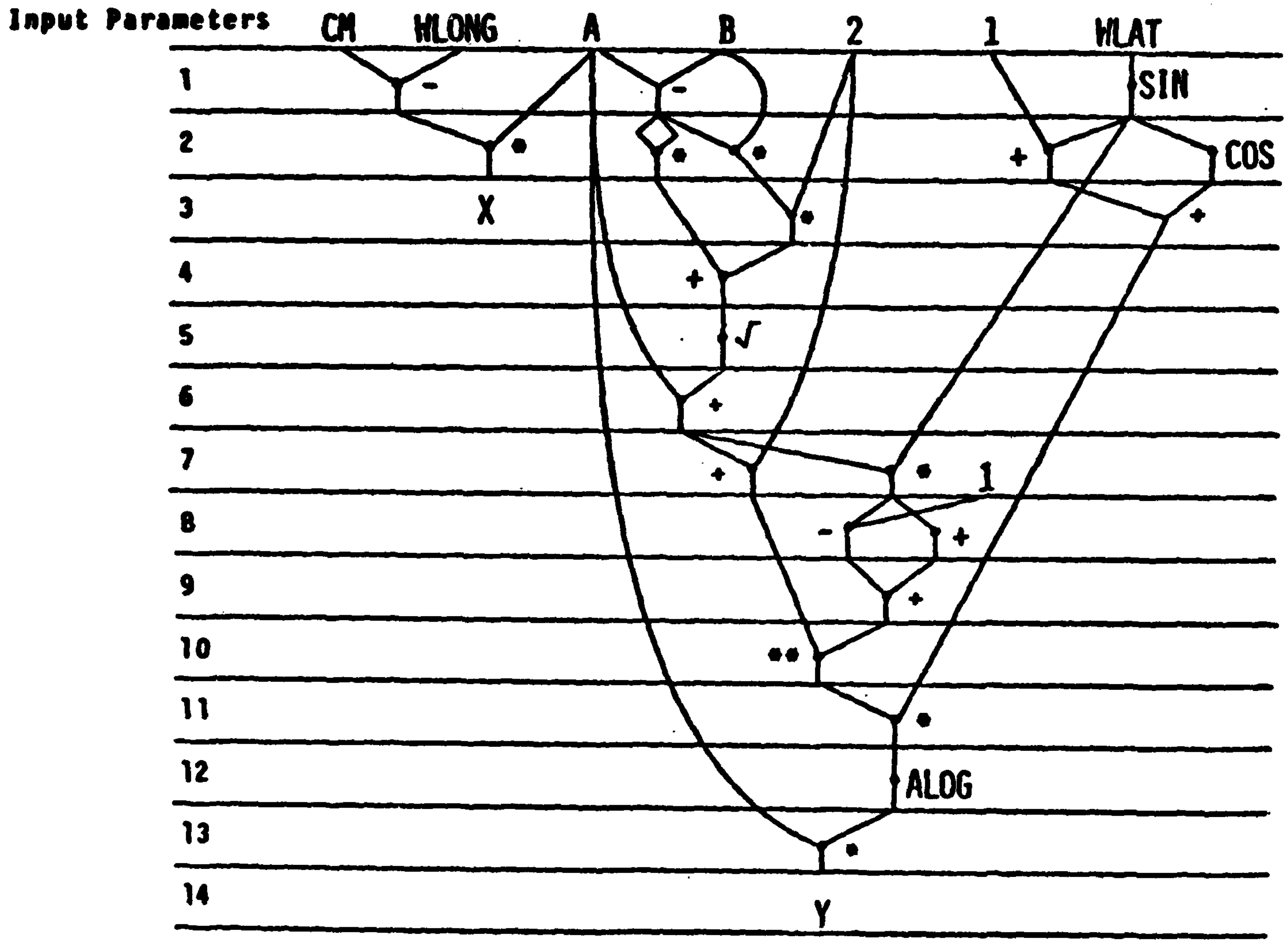


Figure 2.2 Mercator Data-Flow Graph

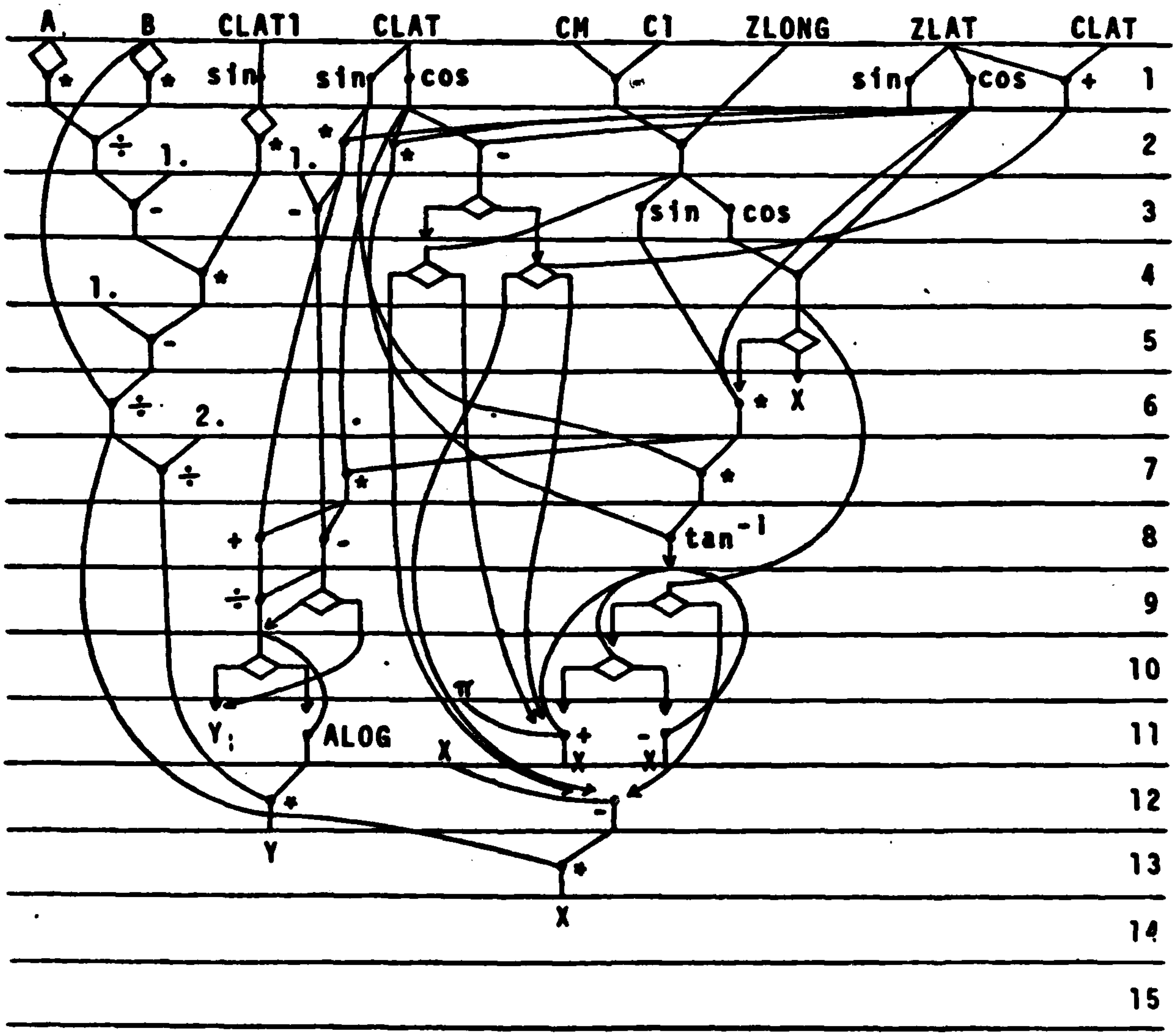


Figure 2.3. OBMERP Data-Flow Graph

While data-flow graphs illustrate the concurrency of a particular arithmetic process, the number and types of unique arithmetic operations is important for determining the number and types of arithmetic functional units that will actually execute the arithmetic operations. Table 2.2 lists the total number and types of different arithmetic operations for the initialization and for the main routines of each transformation algorithm and inverse.

Hardware units are available for the basic mathematical functions (multiplication, division, addition, and subtraction). These "simple" functions consequently form the backbone of the arithmetic processing unit, of whatever form it may finally take. The remaining "non-simple" functions, on the other hand, cannot be handled as conveniently.

2.1.1.2.1.1 Simple Arithmetic Functions. The basic binary mathematical operations (multiplication, division, addition, and subtraction) are commonly realized in distinct hardware units. The state of the art in performance of these units is continually improving in terms of the number of bits of precision and the speed of execution. The LSI Applications project assumed the use of units with the performance figures listed in Table 2.3, based on state-of-the-art TRW designs.

2.1.1.2.1.2 Non-Simple Arithmetic Functions. While distinct hardware units are available for the simple mathematical functions, the trigonometric, square root, and logarithm functions required by the transformation algorithms pose a special problem. There are three basic techniques for handling these functions:

1. Cordic algorithm,
2. Table lookup, and
3. Polynomial evaluation.

Each of these three techniques has its own set of associated advantages. Implementation, as is usually the case when paper design must be turned into hardware reality, tends to eliminate approaches based on the relative disadvantages of each technique.

Undoubtedly the fastest possible technique, table lookup involves addressing a very large piece of memory to retrieve the appropriate value of the function. Although this technique promises a solution for, as an

Table 2.2. Dataflow Analysis Results

PROG NAME	INIT /MAIN	SQUARE ROOT	INTEGER /POWER	+	−	*	/	SIN	ASIN	COS	TAN	ATAN	LOG	#REAL POWER	# OF STEPS
ALEACO	INIT	2	3	3	3	6	2	2	0	1			1		19
	MAIN	0	1	2	1		0	1	1	0			0		14
ALEAGP	INIT	1	5	4	2	10	2	2	3	2	0				22
	MAIN	0	3	2	2	3	2	1	0	0	1				23
EQARZ	INIT	0	2		1	0	1	0	0						3
	MAIN	1	1		2	2	0	1	1						11
AZEQU				1	1	2	1	1	1	1					10
AZEQZP		1	2	2	2	1	2					1			7
AGNOMO	INIT				1	0	1	1		1					1
	MAIN				1	5	2	0		0					6
AGTOGP	INIT		1	0	0	2	0	1	1						2
	MAIN		2	2	1	2	2	0	0						6
LAMB	INIT	2	1	0	5	4	3	2		2			2		19
	MAIN	2	2	2	2	3	2	2		2			0		13

Table 2.2. Dataflow Analysis Results (Continued)

PROG NAME	INIT /MAIN	SQUARE ROOT	INTEGER POWER	+	-	*	/	SIN	ASIN	COS	TAN	ATAN	LOG	REAL POWER	# OF STEPS
LBTOGP	INIT	2	1	1	2	2	2	2		2	0		1		18
	MAIN	0	0	4	3	4	2	0		0	1		0		25
LAEACP		1	2	1	1	2	2	1							12
LAEAGP		1	2	1	1	1	3								12
MERCTR		1	1	1	2	3	1	1	1				1		12
MRTOGP				1	2	2	2		1						16
OBMERP				1	2	3	1	3		2		1	1		12
OMTOGP				1	1	2	2	1	1	1		1			33
ASTERE		2	2	2	4	2		2		2					18
PSTOGP		1	3	2	2	4	1	2			1				29

Table 2.2. Dataflow Analysis Results (Continued)

PROG NAME	INIT/ MAIN	SQUARE ROOT	INTEGER POWER	+	−	*	/	SIN	ASIN	COS	TAN	ATAN	LOG	REAL POWER	# OF STEPS
POLY	INIT	0	0	3	1	4	4	0		0					10
	MAIN	1	1	1	3	4	1	4		1					11
POLYGP	INIT	0	2	3	1	4	10	0		0					10
	MAIN	1	1	2	3	5	1	4		1					17
UTM	INIT	0	2	2	1	3	8	0		0	0				10
	MAIN	1	2	4	4	8	3	4		2	2				14
UTMGP	INIT	0	2	3	1	3	6	0		0				0	10
	MAIN	1	4	5	3	7	4	4		1				1	17
FWDGP	INIT		8	0	1	1	2	0		0		0			5
	MAIN		3	2	6	5	2	2		2		1			37
XYZFLH		1	2	3	3	2	2		1		1				19
ORTM6A					2	5		3		3					5
LCGO				3	5	8									7
FLHXYZ		1	2	1	1	1	1	1	1	1					10

Table 2.3. Arithmetic Functional Unit Performance

operation	execution time
addition subtraction	200 nanoseconds
multiplication	400
division	550

example, the sine function within approximately 1.4 microseconds, the memory requirements are extremely large, and, in an actual implementation, prohibitive.

The next fastest technique uses a technique called the cordic algorithm. The cordic algorithm would return the result within approximately 2.0 microseconds, but suffers the disadvantage of needing special hardware elements such as barrel shifters to implement it.

The last technique, polynomial evaluation, involves use of the familiar technique of solving a polynomial that approximates the original function to a specified degree of accuracy depending on the number of terms included in the polynomial. For example, the sine function can be approximated by the equation:

```
SIN(Y) = PI/2*Y - Y**3(S(0) + S(1)*Y**2 +
                   S(2)*Y**4 + S(3)*Y**6 + S(4)*Y**8)**2
```

where the S values are known constants.

Unlike the previous two techniques, polynomial evaluation does not require any further hardware augmentation beyond the basic mathematical function hardware units. Examining the equation for the sine function, it becomes evident that it is but a combination of the various "simple" mathematical operations. Consequently, the 5.5 microsecond solution time required with this technique is balanced by the advantage of using the arithmetic functional units already available, rather than involve further implementation of specific hardware elements. This use of the simple arithmetic functional units is made possible by providing a subroutine that executes the polynomial evaluation using these basic functional units.

2.1.1.2.2 <u>Non-Arithmetic Functions</u>. Non-arithmetic functions fall into the arena of device control. The possible paths include:

1. Control of execution of the transformation routines,
2. Control of data movement associated with the transformation routines,
3. Control of status information within the device,

4. Control of status information between the device and the host, and

5. Control of (downloaded) code from the host to the device.

This set of control paths does not pose a serious burden on most processors. These control paths do imply, however, a requirement for the processor to execute conditional branch instructions and to perform logical tests of (status) bits in the device.

2.1.2 Definition of Trade-Off Areas

In the previous section, the discussion focused on the nature of the computational processes involved in implementing an order of magnitude improvement in performance of the execution of the transformation algorithms. In any real hardware system, one choice of possible implementation paths causes restrictions in the use of certain other possible implementation strategies. The relative effect of the restrictions must be examined in order to plot the correct course to an implementation that best meets the performance goals in the most cost-effective manner.

2.1.2.1 Data Transfer

Data transfer, which includes input points to the device for transformation, output transformed points, and (downloaded) microcode, is a critical path in the total system throughput.

2.1.2.1.1 Response. Response is defined as the time between transmittal of data to be transformed and the reception of the transformed data from the device. While the device may deliver a single transformed point (composed of two values) at a rate ten times faster than the current software implementation, it may not be in the best interests of system throughput to handle data point by point. Each bus access requires execution of a bus protocol and the cost of the accompanying overhead. Thus, while a single point could be returned quickly, the extra bus access would actually deliver a lower system throughput than if the transformed data were returned within a burst of points, which would have the associated overhead of only a single bus access protocol.

2.1.2.1.2 Throughput. Throughput is defined as the rate over some specified time period at which data are moved from the host to the device,

transformed, and moved from the device back to the host. For reasons of bus contention mentioned in the previous section, it is likely that the throughput of the system could be increased at a slight cost to response. Thus if a thousand points were transferred as a block to the device for transformation and returned as a block of a thousand transformed points, a simple analysis indicates that response would be about a thousand times slower for a single point in the block than the response if the point were sent, transformed, and returned individually. On the other hand, system throughput would be greatly enhanced since 4000 individual bus accesses (each point consists of four words) would be reduced to a single access for the entire block. Although the operating environment has not been indicated, it is reasonable to assume from the nature of the transformations themselves that response is less stringent a requirement than total system throughput. Consequently, the proposed device should provide a means for transferring a block of data to and from the device. For purposes of increased single-point response, the proposed device should not exclude the possibility of transferring a single point at a time.

2.1.2.2 Processing

Since the intent of the proposed design is to optimize the architecture of the device for the execution of the DMA transformation algorithms while maintaining flexibility, the nature of the proposed architecture is heavily dependent on the characteristics of the algorithms themselves. Paragraph 2.1.1.2.1 provided a detailed description of the nature of the algorithms through a device called a data-flow graph. These data-flow graphs explicitly contain the number and types of different operations required to execute the transformations. Clearly arithmetic operations are the most commonly-used functions and are the processing elements most deserving careful study in order to meet the tenfold performance improvement goal.

2.1.2.2.1 Arithmetic Functions. Since the clear majority of arithmetic operations used in execution of the transformation algorithms are the "simple" mathematical operations, it is appropriate to optimize the execution of these simple mathematical operations.

2.1.2.2.1.1 Simple Arithmetic Functions. Not a great deal of trade-off freedom is available in the implementation of the "simple" mathematical functions because of their importance in the successful implementation of a device with the required speed of execution. Table 2.3 lists the expected speed of these hardware units based on TRW state-of-the-art designs.

2.1.2.2.1.2 Non-Simple Arithmetic Functions. Paragraph 2.1.1.2.1.2 described three possible approaches to the non-simple functions. While the cordic algorithm provides a fast execution, the special-purpose hardware over and above that required for the "simple" functions makes this approach less attractive. On the other hand, the table lookup method provides an even faster execution speed. Unfortunately, the memory required for this implementation becomes excessive as resolution between points in the table is made finer.

The technique providing the most amenable hardware solution within the performance constraints is polynomial evaluation. This approach would involve reducing the appropriate polynomial to a subroutine which would call the "simple" mathematical hardware units to execute the processing. This use of the existing simple hardware units has one distinct advantage associated with this technique: it becomes even more worthwhile to concentrate on optimizing the performance of these "simple" mathematical hardware units.

2.1.2.2.2 Non-Arithmetic Functions. The control paths shown in paragraph 2.1.1.2.2 are the source for the expected non-arithmetic functions. They decrease in importance relative to the specified goals of the project as one descends through the list. For the most part, the control function requirements are basic to the operation of the device and consequently do not permit a great deal of tradeoff freedom.

The possibility of downline loading microcode is somewhat different however. Downline loading has the effect of requiring a separate path which would differentiate between data for transformation and code for storage and subsequent execution. Otherwise, downloading does not introduce a significant increase in non-arithmetic operations.

2.1.2.2.3 Algorithm Characteristics. The mathematical operations contained in the execution of the transformation algorithms have been listed in Table 2.2. The other aspect of the characterization of the algorithms is

the degree of concurrency that may exist between functional subsections of the same algorithm.

The data-flow graphs shown as examples in Figures 2.2 and 2.3 illustrate the degree of concurrency in these algorithms. It is evident from these data-flow graphs that the concurrency in execution of subsections of the algorithms does not follow a great deal of regular order. Consequently, implementation of parallel processing of the algorithms would require special logic to recognize when a subsection could be processed based on the state of the data in other parts of the device (or at other stages of execution within the device).

2.1.3 Tradeoff Recommendations

While this set of design principles discussed in the preceding paragraphs is based on certain assumptions concerning the DMA operating environment, their adoption does present certain other advantages, both in implementation of the device and in execution of the transformation algorithms. Thus trading throughput for single-point response will provide not only an increased system throughput for transformation data, but will also provide an increase in the overall throughput of the host system.

Similarly, while adoption of the polynomial evaluation technique for implementation of the "non-simple" arithmetic functions does result in a somewhat slower execution of these functions, it does improve the duty cycle of the "simple" hardware units, thus lending further impetus to optimizing their design (which would further optimize device performance).

2.2 ARCHITECTURES

Concurrent with the algorithm analysis was the analysis of various candidate architectures for implementing the desired solution to the stated problem. Several possible architectures exist which are capable of high-speed computation of the type required; however, a key performance criterion is the speed of the various mathematical functions since it has been shown previously that the execution time of the mathematical functions drives the speed of execution of the transformation algorithms. Table 2.4 illustrates the various speeds of execution of mathematical operations in several of the candidate architectures.

Using the speed of execution of the mathematical operations on the host machines as the performance baseline, it is clear that microprocessor-based

Table 2.4. Timing Summary of Mathematical Operations on Several Candidate Architectures

FUNCTION	CSIP ARRAY PROCESSOR	INTEL 8086	INTEL 432	FLOATING ARRAY PROCESSOR	PROPOSED DESIGN	PDP 11/60	ECLIPSE C300
ADD	0.36	14			0.02	5.44	4.2
MUL	0.35	18	6	.167	0.35	5.66	7.4
DIV	1.50	39		3.83	0.55	8.66	8.6
SIN	2.4			4.9	C – 2 T – 1.4 P – 5.5	76.77	
SQRT	6.3	36		3.83	C – 3	61.16	
TAN		110			C – 2.5 T – 1.4 P – 5.5		
ALL TIMES ARE IN MICRO SECONDS C=CORDIC. T=TABLE LOOK UP. P=POLYNOMIAL EXPRESSION							

designs do not possess the requied speed of execution to meet the performance level required. Array processors, on the other hand, just barely meet the required speed of execution of the basic mathematical functions. The clear performance leaders are the hardwired and microprogrammed architectures.

A microprogrammed circuitry approach was selected over hardwired approaches for the sake of flexibility, as will be shown in the subsequent paragraphs.

2.2.1 Array Processors

The primary feature of array processors is their ability to handle a high degree of parallelism of mathematical operations. Multiple data paths and multiple arithmetic functional units expedite vector-type processing.

Implementation of the proposed device by means of this architecture would include an instruction/data fetch unit, data conditioning units for pre- and post-conditioning, an address-generation unit, a control unit, and multiple arithmetic units of the various required types. The units are essential stages for most computational tasks. Array-processing architectures would perform these operations as well as other operations concurrently.

The pipelining and parallelism available with this architecture would allow simultaneous fetching of the instruction and operations of the data on every cycle, providing further improvements in performance. Multiple data paths, one of the major features of array processors, would provide the high degree of concurrent operation expected from array processors; however, programming coordinated movements through these multiple data paths would prove to be a complex task due to the irregular nature of the transformation algorithms. The architecture is more suited to movement of highly regular data structures, such as arrays, all of whose elements require the same mathematical operation. Since the algorithms deviate from this highly-ordered structure, the programming of data movements and the coordination of mathematical operations at the various nodes would dramatically increase in complexity.

Finally, available array processors do not provide the level of performance that is required, as is shown in Table 2.4.

2.2.2 Data-Flow Machines

At the opposite end of the processing spectrum, data-flow machines are well suited to processing algorithms with irregular structure. Data-flow machines are essentially self-timed systems where sequential "clocking" of the various stages of the hardware depends solely on the rate at which input data reach the next stage of operation. Since organization of the data is immaterial, this architecture is well suited to processing those algorithms whose data structures are other than arrays, although data-flow architectures handle arrays as well as any other arbitrary data structure.

Several serious drawbacks make this approach less attractive, however. Since data-flow architecture is a relatively immature design approach, the relative lack of a disciplined approach to their design would introduce problems during the design phase. Similarly, decomposition of the algorithms and assignment of sections of the algorithms to portions of the data-flow machine could easily fail to make optimal use of the self-timed features of this approach. Furthermore, the data-dependent nature of the architecture would limit the flexibility of a particular implementation, perhaps to the extent that each algorithm would require a separate data-flow machine.

These problems led to the decision to abandon a data-flow architecture despite its inherent suitability to processing semi-regular algorithms such as the transformation algorithms.

2.2.3 Hardwired Architecture

Undoubtedly the architecture promising the highest performance, the hardwired circuit approach suffers from the serious problems of lack of flexibility, difficulty in maintenance, difficulty in modification, high power requirements, and large size.

Figure 2.4 illustrates an example of a hardwired circuit. As noted in this figure, the example is for but a single algorithm. Consequently, 47 other circuits would need to be constructed, each with its own maintenance requirements.

Since the algorithm would be directly mapped into hardware components, any change in process flow in the algorithm that might later be introduced would require the removal of the unit from service and a major redesign effort to place the modification into the existing circuit.

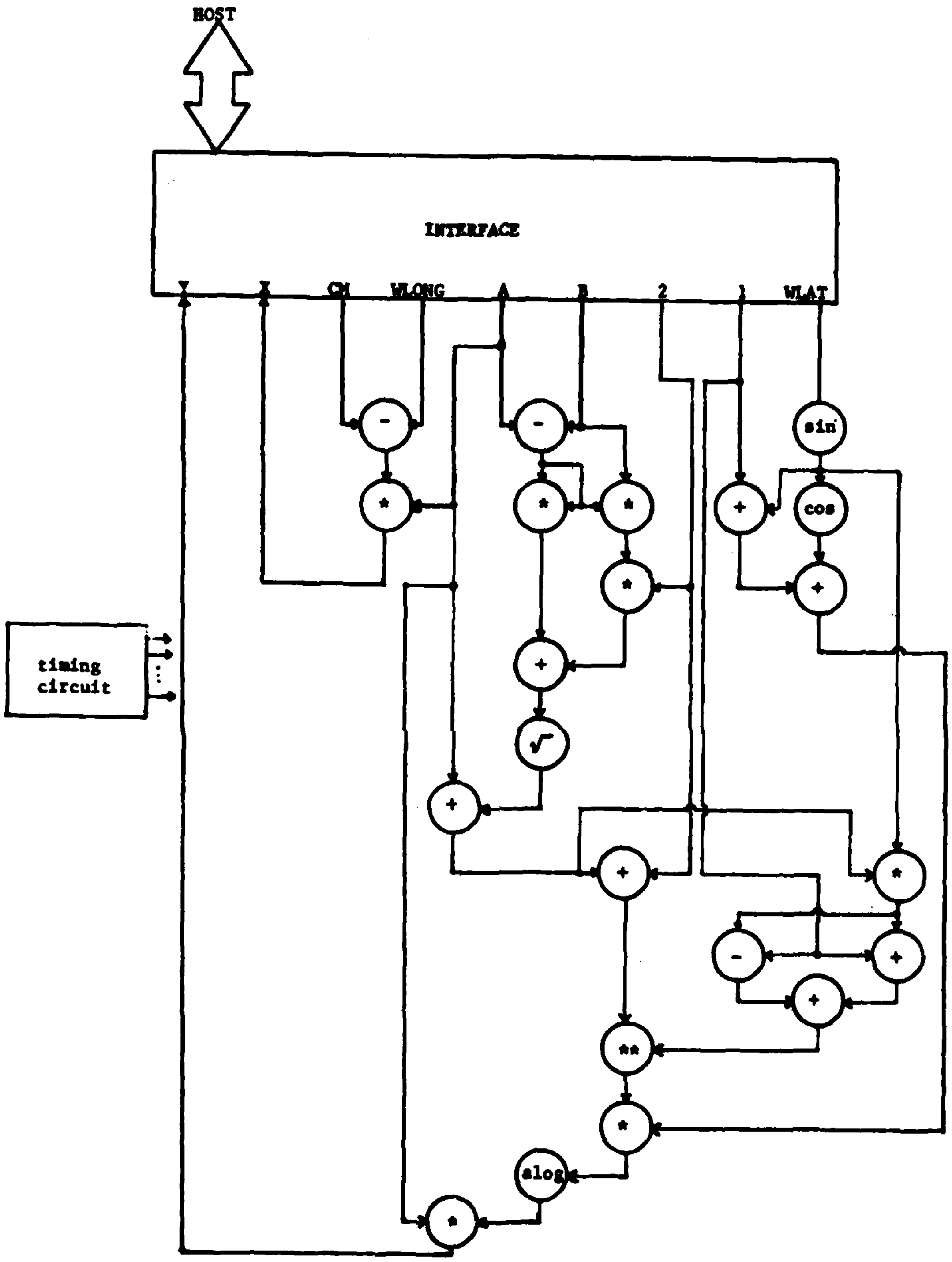


Figure 2.4 Hardwired Implementation of Transformation Algorithm Mercator

Furthermore, since hardwired circuits would be composed of discrete small-scale integrated circuits, the resulting circuit would be left with high power requirements as well as occupying a large area of space either in a separate cabinet or within the minicomputer cabinet.

Primarily due to the need to construct and maintain a separate circuit for each algorithm and inverse, this approach was abandoned.

2.2.4 Microprocessor-Based Architectures

Implementing the proposed device using microprocessors would provide a structure such as that shown in Figure 2.5. This design would consist of a 16-bit general-purpose microprocessor, such as the Intel 8086, a numeric co-processor, such as the Intel 8087, local memory, and circuitry to interface with the host computers.

The primary advantage of this approach is the simplicity of the design and implementation. The fatal disadvantage, however, is the very slow speed provided by the components as previously shown in Table 2.4.

2.2.5 Microprogrammed Architecture

As shown in Table 2.4, the only architecture providing the required performance, other than the highly-effective hardwired approach, is the microprogrammed architecture. Microprogrammed circuits generally consist of a limited set of very high speed digital components, including:

1. Memory, for storing the controlling microprogram;
2. A controller, for sequencing through the control program and issuing control signals to the rest of the device;
3. Interface circuit, to provide connection with a host computer, and
4. Functional circuitry, to implement the desired function of the device.

In particular, the major components identified in the course of this project include:

1. A microprogram control unit (MCU),
2. An arithmetic processing unit (APU), and
3. An interface control unit (ICU).

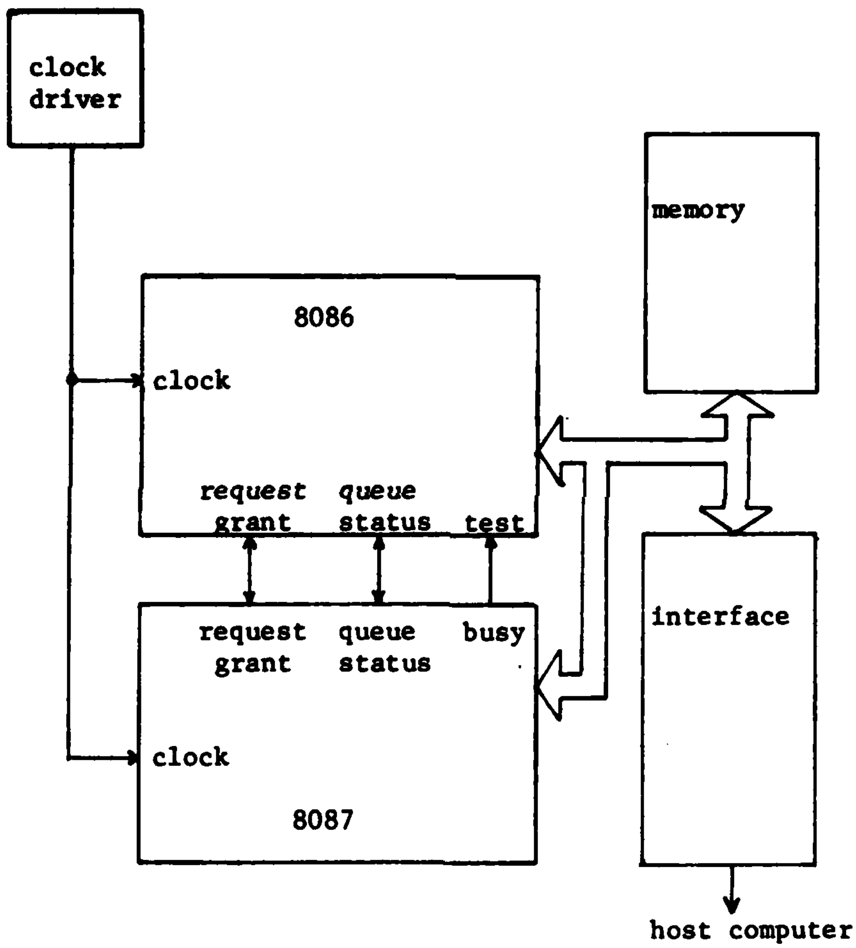


Figure 2.5. Microprocessor-Based Architecture

Figure 2.6 shows the major component level design of the proposed device based on this approach.

Identification of global functions and their assignment to particular units within the overall system is a relatively straightforward task.

The selection of the particular details within each unit and the selection of the number of units is a highly involved problem that must consider:

1. The required performance level of the overall device;
2. The peculiarities of the operations to be performed on the device; and
3. The partitioning of subsections of the algorithms to parts of the device, particularly when more than one type of each unit is available.

The required performance of the device has been discussed in paragraph 2.1.1.1.2. The nature of the algorithms has been discussed in paragraph 2.1.1.2.1. The internal details of function of the major components, and the considerations for selecting the appropriate number of each unit will be discussed in the following paragraphs.

2.2.5.1 Automated Firmware Design

Although a microprogrammed architecture possesses sufficient generality to execute a wide range of algorithms, it is possible to optimize the architecture through correct choice of the number of components based on the particular nature of the algorithms. A powerful set of tools, called Automated Firmware Design (AFD) aids, is available at TRW to accomplish this algorithm-dependent design trade-off.

Using a "standard" form of a microprogrammed unit similar to that described above, AFD analyzes a specific algorithm and fills in the details in the standard form configuration, including optimal number of functional units. AFD provides a significant increase in productivity to designers by simultaneously producing the actual microcode needed to execute the target algorithm on the proposed configuration of hardware elements.

The AFD process then, shown in Figure 2.7, becomes a key performer in the design and development phase as a design cycles iteratively through evaluation and alteration to a finished product.

The designer uses this powerful tool by providing, in addition to an initial hardware design, the algorithm itself written in a structured version

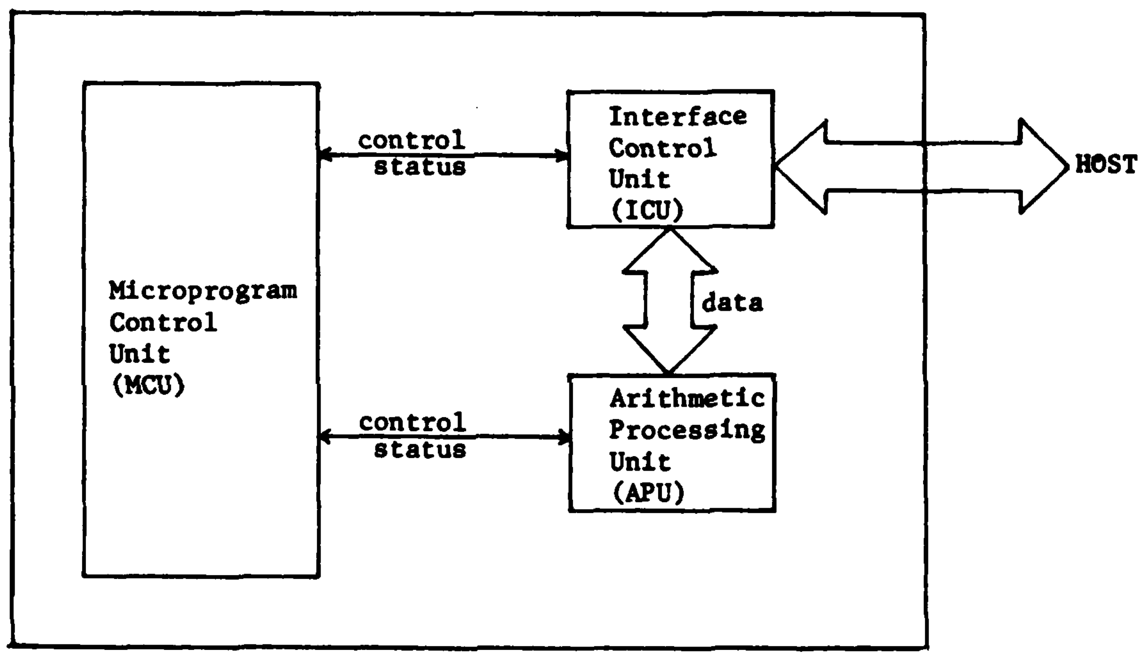


Figure 2.6. Microprogrammed Architecture

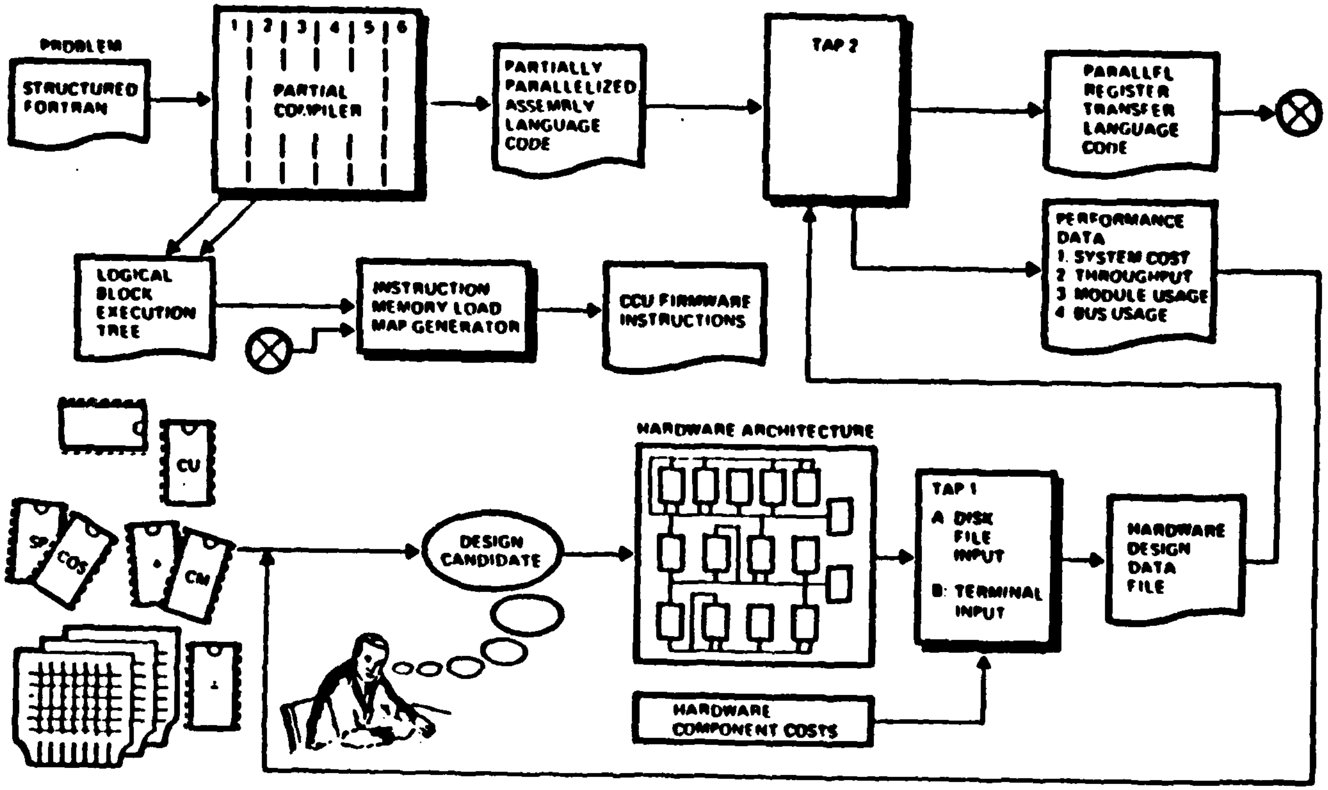


Figure 2.7. Automated Firmware Design (AFD) Process

of FORTRAN. Passing this form of the algorithm through a special AFD compiler reduces the problem to a series of sequential blocks of operation. Each block is internally divided into groups of single operations (such as the multiplication of two values) called "precedence levels". Given unlimited hardware resources, all the operations within any single precedence level could be executed in parallel.

The next phase in the AFD process involves entering the blocks into a routine called TAP-2, Throughput Analysis Program. TAP-2 optimizes resource allocation and scheduling, collects performance data, and generates parallel microcode. TAP-2 optimizes resource allocation and scheduling by reducing the operations to the fastest possible sequence of register transfers. The performance data provided by TAP-2 is a particularly useful element in the design cycle. By producing an analysis of the function of the algorithm on a particular hardware configuration, TAP-2 furnishes feedback that enables the designer to modify the candidate hardware configuration.

The "parallel register transfer code" noted in Figure 2.7 is the microcode output from TAP-2 that controls execution of the input algorithm by the standard central controller unit used in all AFD target hardware designs.

TAP-1 combines the adjusted hardware configuration with performance and cost data to produce a new set of inputs for TAP-2. Thus the target hardware design iteratively passes through a series of modifications until an optimal configuration has been realized.

The actual firmware microinstructions are finally produced at this point by combining the output from the compiler with the parallel register transfer code. This occurs in the AFD process called the instruction-memory load-map generator.

TAP-2, then, is a mechanism for obtaining the optimal hardware configuration (subject to the constraints of the "standard form" design) based on a specified algorithm. This process provides two important design aids:

1. Iterative optimization of the number of functional units, and
2. Production of the final microcode for driving the controller.

2.2.5.2 LSI/VLSI Design

The state of the art of hardware design is currently moving toward implementing large regular architectures as single chips, or at most a small set of chips. This VLSI design will eventually complement such design aids as AFD by directly producing a chip based on specifications provided by design tools like AFD. At the moment, however, very large integrated circuit design is a relatively high risk technology and design must still pass through a series of steps not dissimilar from those used in the past for large scale integrated circuit design. In addition, the process technologies commonly used in VLSI, such as nMOS, do not provide the high speed performance level that LSI and MSI circuits using different process technologies currently provide. Consequently, LSI and MSI are expected to be the scale of integration that will likely be used in implementing the proposed design.

Five major levels of design are likely to be used in implementing the proposed device using large scale integrated circuit technology. They include:

1. Architecture design,
2. Logical design,
3. Physical design,
4. Mask design, and
5. Mask and chip fabrication.

Figure 2.8 shows how these major steps are related for chip development. Descending from the top architectural level down to the mask and fabrication level, each design level represents an increasingly more detailed description of the design. In essence, LSI design is a step-wise refinement from a high-level specification to a final mask set.

Each level shown in Figure 2.8 is characterized by the four major design activities illustrated in Figure 2.9:

1. Synthesis,
2. Description,
3. Evaluation, and
4. Validation.

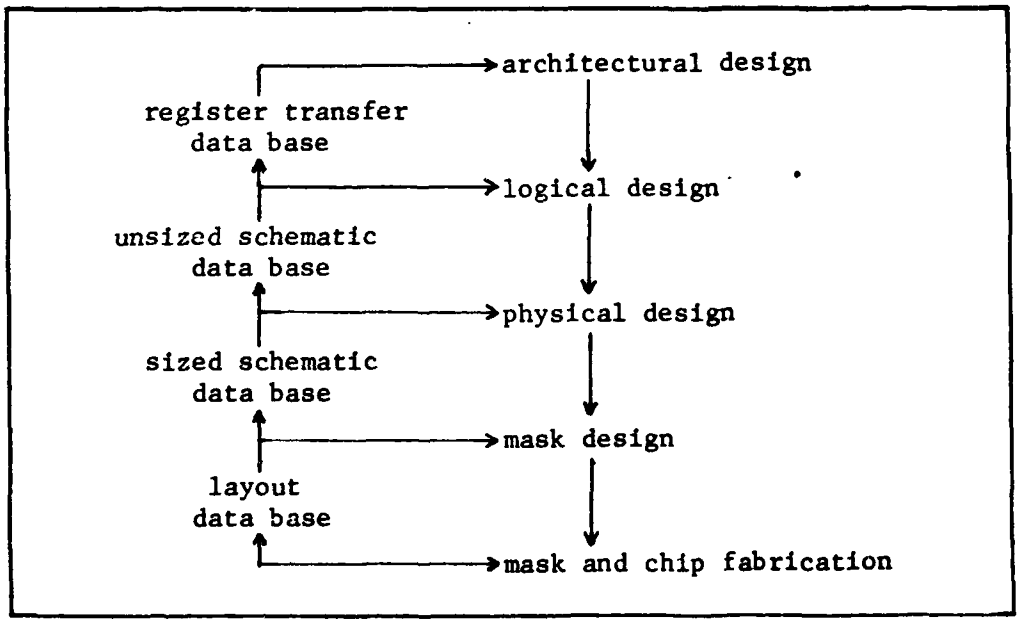


Figure 2.8. Steps in Large-Scale Integrated Circuit Design

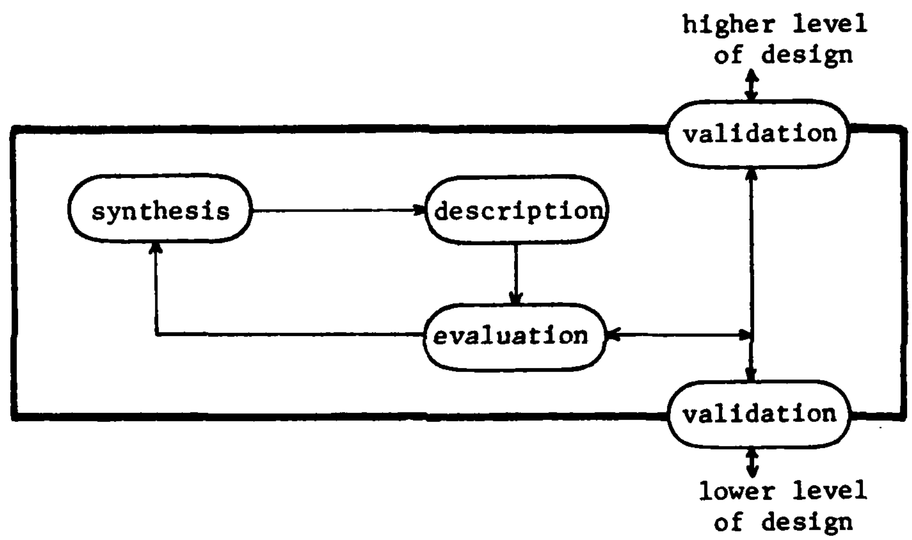


Figure 2.9. Detailed Flow of Chip-Design Activities

Synthesis, the transformation of one level of description to the next level precedes the next step of description. Each level has its own language for describing a design at that level. After synthesis and description, the design is evaluated against the requirements and constraints placed upon it. For an LSI design, the major constraints are die size, speed, and power. These are reevaluated at each level, where often a design is forced to return to a higher level when a constraint cannot be met at a lower level. A design is also evaluated for behavioral and structural completeness, by no means a trivial task, even after a rigorously-constructed description has been shown to be free of such errors as unspecified states or unconnected gates. Development of computer tools, including simulators at three different levels is required for evaluation of the design. It is normally helpful to drive these tools directly from the corresponding design description. Automatic translation from the reference descriptions to the input formats will be required to expedite this evaluation process.

The final step is validation and verification. Validation is a semi-formal process whereby the design at a particular level is shown to be equivalent in either behavior or structure to one or more higher-level designs. Validation involves using test cases or worst-case models that attempt to stress the design to its logical and physical limits. In contrast, verification is a strictly formal process for demonstrating the equivalence of two design levels under all specified situations. Consequently, the validation will be used to show behavioral equivalence between two levels, while verification will be used to show their structural equivalance.

A design data base will be created to save all computer descriptions, models, and tools at each level. This data base will be especially important because it represents the complete design at any level and is the means for communicating the design to the next lower level.

2.2.5.2.1 Architectural Design Level. The architectural level is concerned with high-level system organization. The level of design will then be decomposed further into microarchitectural levels, reflecting the fact that the microprogramming technique is the key implementation technique for the LSI device. The design description at the architectural level will be in the form of a SMITE description containing a set of chip models, providing a framework upon which the design is tested. The user interface available with SMITE

Application Support Software (SASS) allows a user to manipulate the emulation produced by SMITE and query the system about the state and contents of values local to each model. Section 2.3 below discusses details of emulation in more detail.

System components such as memories and I/O interfaces are also represented in the emulation so that the complete system can be modeled. The constructs of this class of programs will also allow generation of multiple copies of a given chip model. This facilitates modeling systems with multiple copies of the same component.

As discussed below, SMITE is an extremely valuable tool at the architectural level. In addition to providing a means for formally defining and evaluating the architectural design, it provides the reference description for validation and verification of subsequent design levels.

2.2.5.2.2 Logic Design Level. The next level is the logic design level. This level considers the logic structure required to implement the architectural design. At the logic-design level, a more detailed SMITE design description will be considered as a set of unsized logic schematics. Unsized in this case refers to a network of logic gates devoid of physical circuit information. For evaluation, emulation under SASS will simulate logic design at the gate level.

With respect to the architectural design, the logic design will be validated in the following way. First, a set of high-level test programs will be developed and executed under SASS. This will allow the designer to trace the input stimulus and output response on a cycle-by-cycle basis. Next, a more detailed SMITE description will provide data at the logic-gate level. Checks will verify the correct translation from high-level SMITE descriptions to lower-level detailed descriptions.

2.2.5.2.3 Physical Design Level. The physical-design level considers the physical laws that govern the behavior of microelectronic circuits. At this level, the unsized schematics of the logic-design level are transformed into sized schematics. Standard logic gates often receive only transistor-sizing information, whereas memory arrays or other structures usually need to be expanded to show the necessary details of timing generators, sense amplifiers, and buffers. The device sizing will require the use of a circuit simulation program such as SPICE. An accurate device model and an accurate circuit

description are essential to meet both the circuit function goals and the circuit performance constraints.

In general, the circuit design will be correct if the logic is not changed when it is sized. If the logic is altered, then this change will be passed back up through the design chain.

It is important to note that due to the time and computer resources required, it may not be practical to simulate the entire chip described by the sized-schematic data base. Instead, subcircuit analysis techniques make the circuit simulation manageable.

2.2.5.2.4 Mask Design Level. During the mask-level design for the different elements of the LSI device, the sized schematics for the corresponding elements are turned into the geometric shapes to be reproduced as different masks during the fabrication process. At this level, the chip is drawn as a composite layout, as detailed by the sized schematic. The description language at this level will be one of geometric shapes and shape placement, such as CIF or the higher level LAP. The data base is an online composite in a computer graphic system.

The composite will be correct if the shapes are drawn and placed correctly. Two issues will determine this correctness. First, each semiconductor technology has a specific set of design rules that must not be violated. Creation of a design-rule checking program is the usual means to verify observance of the design rules. Second, the composite must correctly implement the sized schematics. At this point a connectivity-verification program will be required to verify the structural equivalence of the sized schematics and mask design. The program will create a network list from the sized schematics and a network list from the design data base. The connectivity-checking program will compare the two network lists to verify the structural correctness of the layout and the integrity of the graphics data base.

The importance of error-free layout at the LSI level of complexity cannot be overstated. To ensure that the mask design meets the die-size constraints, topographical considerations are reviewed at each design level. A chip plan is developed early during the microarchitectural design, beginning as a coarse block layout that will be refined to a detailed interconnected "road map". A copy of the chip plan resident in a graphics system will allow mask designers to share the layout and files.

2.2.5.2.5 Mask and Chip Fabrication Level. At this level, the design is described on the silicon chip itself. After masking and fabrication, the first task will be to test and validate the hardware implementation. Throughout the design cycle, the SMITE description of the component design will be the reference to which all other levels of design will be compared.

Automated testing of the chip will rely on a test pattern developed from the stimulus and response data generated by the SMITE description emulation.

2.2.5.3 Controller Unit

The design of the controller unit for microprogrammed circuitry is not particularly given to design trade-off considerations. In general, such a controller unit should include:

1. High speed memory for storing the controller microprogram,
2. High speed sequencer for executing the controlling microprogram by stepping through the high speed memory,
3. High speed controller for decoding the microprogram and distributing the appropriate control signals throughout the device,
4. High speed clock for ensuring synchronized sequencing through the instructions.

2.2.5.4 Arithmetic Processing Unit

The single source of all arithmetic processing, this unit must be designed to optimally deal with the particular nature of the target algorithms. In particular, the arithmetic processing unit must contain sufficient hardware resources to execute the mathematical operations at the required rate. The determination of the correct number and types of arithmetic functional units is based on analysis of the algorithms and consideration of the results of the analysis with respect to the required performance of the mathematical processing.

2.2.5.4.1 Types of Arithmetic Functional Units. The analysis study reported in Paragraph 2.1.1.2.1 suggests a breakdown of all mathematical operations into two classes:

1. Simple (addition, subtraction, multiplication, division) and

2. Non-simple (trigonometric, square root, logarithm, ...).

Furthermore, the analysis in Paragraph 2.1.1.2.1 demonstrates that the optimal allocation of function into hardware elements involves using hardware elements for the simple class of mathematical operations only. Using a microcoded subroutine to solve the polynomial expansion of the required non-simple mathematical functions provides the optimal match of hardware configuration with desired performance.

2.2.5.4.2 Number of Arithmetic Functional Units. The automated firmware design tools described in Paragraph 2.2.5.1 are undoubtedly the most powerful mechanism for determining the optimal number of arithmetic functional units in the arithmetic processing unit. However, Table 2.5 lists the expected performance of the proposed device when only a single instance of each type of arithmetic functional unit is used in the processing unit. A further point illustrated in this table is the ability for simple serial processing of the algorithm to effect a performance level that more than adequately meets the required level. Consequently, in the interests of meeting the stated requirements in the most cost-effective manner, simple serial processing with a single instance of each type of arithmetic functional unit is considered the most effective approach.

2.2.5.5 Interface Control Unit

Careful design of the interface unit could provide a long-term benefit. By localizing all the host-dependent features of the proposed design in the interface unit, the designer will find eventual attachment of the device to different host computers a relatively straightforward task. The interface unit, then, should include:

1. The capability to translate data between the format of the host computer and that of the proposed device's arithmetic processing unit, and

Table 2.5. Algorithm Timing (Cost of Transforming a Single Point Using Simplest Architecture)

ALGORITHM (1)	PDP 11/60 (EMULATION)		DEVICE (ESTIMATED) (2)	
	INIT	MAIN	INIT	MAIN
MERCATOR	1180 μSEC	1060 μSEC	7 μSEC	30 μSEC
INVERSE	770	700	3	30
TRANSVERSE MERCATOR	2700	2190	52	79
INVERSE	--	4470	92	180
POLYCONIC	2300	1770	10	70
INVERSE (3)	11830	9560	20	70
LAMBERT	7200	1910	95	75
INVERSE	5800	1160	45	30

Notes:

1) The four algorithms listed account for about 90% of the expected processing load.
2) The times listed do not include the 7 nsec. overhead for the host bus access protocol, nor other host processor overhead costs. ,Also times are for worst-case serial processing of algorithms and assume use of polynomial expression for complex functions.

2. The capability to provide all logic, handshaking, and electrical connections with the host computer's bus, ideally through a standard peripheral interface board attached to the host computer bus.

In order to completely localize host-dependent functions within the interface control unit, the ICU should be capable of a certain degree of processing independent of the microprogram control unit (MCU). The ICU should be capable of performing data translation between formats independently. If the MCU is designed to control this activity, the MCU will not then be totally host independent.

2.3 EMULATION

In the cycle of software development, implementation of an initial design and subsequent alterations are an end in themselves. Hardware design, however, adds an extra step. After each alteration in design, the changes in hardware design must be incorporated into the hardware circuit. This introduces a serious penalty in time and cost for hardware changes.

The solution to this problem is to make use of the same advantages as software development enjoys, by shaking down a hardware design by first manipulating a software description of it that accurately reflects the individual components of the hardware. In addition, the software description of the hardware circuit must be able to be tested dynamically. That is, the timing interaction between the various components must be exercised.

The TRW-developed SMITE hardware description language provides a detailed description of digital hardware at the bit level. Compiling the code on the SMITE compiler and executing the result on the hardware-emulation machine, the Nanodata QM-1, provides a mechanism for testing the design dynamically. Furthermore, a performance-monitoring package SASS, SMITE Applications Support Software, allows the relative effectiveness of changes to the basic SMITE description of a particular hardware circuit to be assessed quantitatively.

2.3.1 Design and Development Support Tool

As a hardware-development tool, there is no single more powerful method available to test performance of a proposed hardware design than hardware

emulation. In addition to providing basic information concerning the features of the hardware design itself, the emulation provides a means to test the proposed hardware within some pre-existing or planned structure. For example, an emulated computer could be used to study the interaction of that computer's hardware with some peripheral device, as the PDP11/60 and the device studied in this project were emulated.

Figure 2.10 illustrates the position emulation holds in the development cycle. Beginning with an initial proposal for a hardware design, a SMITE description of the hardware design serves two major functions:

1. Nails down loose ends in the hardware design, and

2. Provides the input for a test of the dynamic nature of the device.

By executing the compiled description on the QM-1 under SASS, the designer is able to manipulate the operating environment of the hardware through SASS and observe the effects. This enables the designer to identify critical performance parameters of the design and possible associated problems. Modifying the hardware to account for these problems then becomes a matter of modifying the corresponding SMITE description, rather than altering a physical circuit. The cycle proceeds until all the critical performance areas are identified and the circuit is found to handle them all properly.

2.3.2 Features of an Effective Emulation System

As mentioned above, the development cycle, whether for software or hardware, is one of continual cycling through evaluation and redesign. For a technique to be an effective development tool, certain features become necessary elements.

2.3.2.1 Timing-level Description

In order to test any hardware circuit, the description must be able to reach down to the level of the fundamental system clock that would drive the actual hardware circuit. Similarly, for ease of use of the tool, this level of detail should not always be required. If, for example, a study

EMULATION: How is it used?

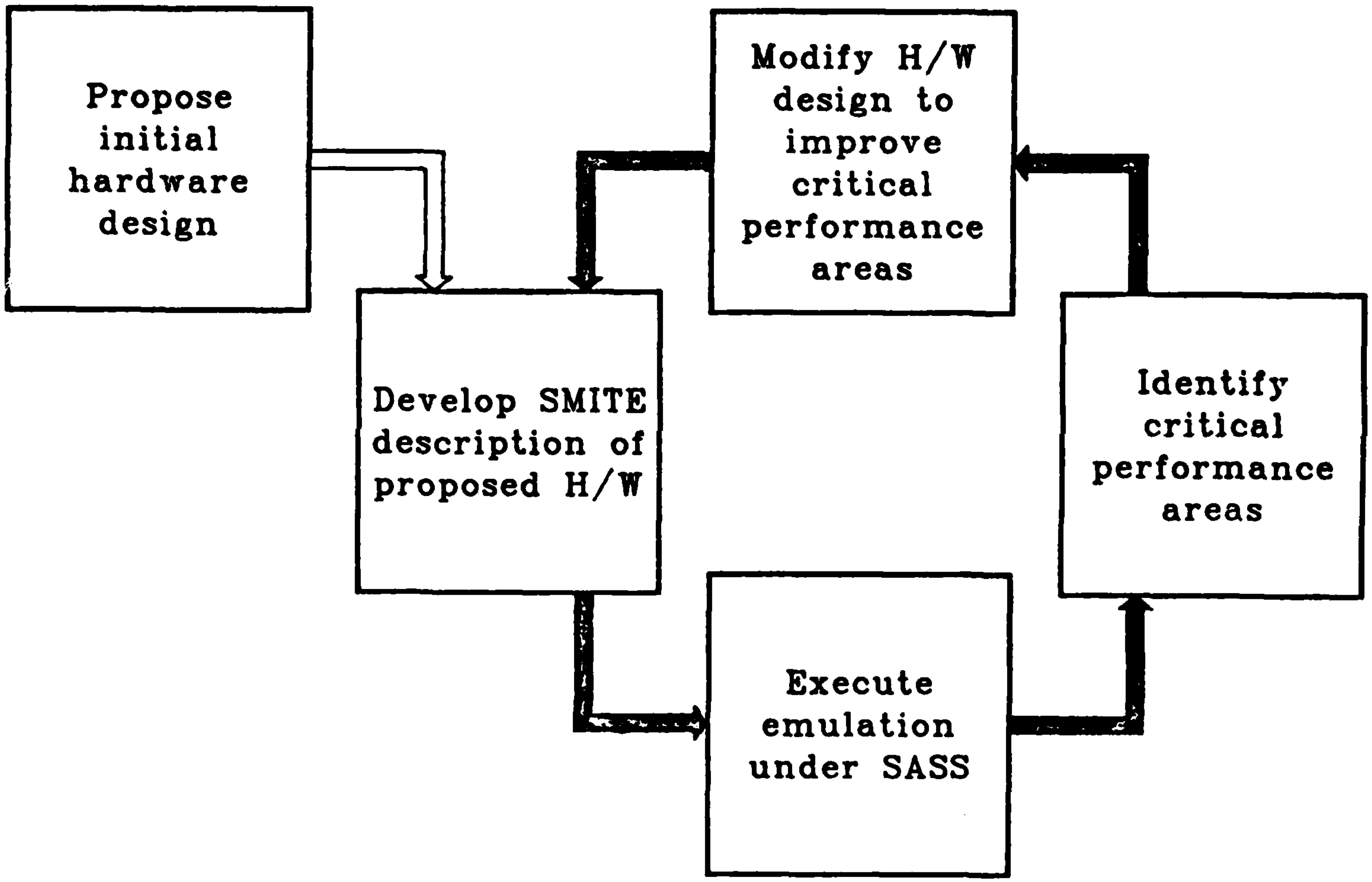


Figure 2.10. Evaluation In the Design Cycle

of bus contention were being undertaken, it would be appropriate to describe the elements directly attached to the bus at the level of the separate phases of the system clock; however, it would be inappropriate to describe the peripheral elements at this detail.

2.3.2.2 Circuit-description Modification

Since the development cycle will repeatedly demand alterations of the design description, the ability to easily alter the description becomes a major point. Ideally, the description language should be a high-order language whose elements themselves are clear enough so that the resulting hardware description may be used directly as a device specification. This ensures a mathematically-"complete" description, since incomplete areas would be identified as soon as an attempt was made to execute the description as an emulation.

2.3.2.3 State Display and Modification

An important attribute of a debugging tool is the ability to both see and change selected variables, or even the flow, of the emulated process. Without this feature, the designer is no more than an observer of the process. The SMITE Applications Support Software (SASS) provides a high degree of flexibility in specifying displays of relevant state variables, and in altering their contents.

2.3.2.4 Breakpoint Processing

Similar to the need for display and modification of state variables, breakpoint processing (the processing of a specified, limited sequence of instructions) is a critical element in a successful development tool. The designer utilizes breakpoint processing by specifying a limit beyond which continued processing of the emulation is not to proceed. Without breakpoint processing, the designer ceases to be even an observer of emulated processes occurring at microsecond intervals.

2.3.2.5 Design Verification

The previous paragraphs merely describe the nuts and bolts holding together an overall framework of an effective design and development tool.

The real strength lies in the ability to emulate the circuit description of an actual machine, thus providing a means for verifying the design. SMITE descriptions are run on the Nanodata QM-1, so that the design under study may be verified against independently-developed software units.

Thus, the emulated PDP11/60 (the executing SMITE description of the PDP11/60 on the QM-1) was tested for proper function by executing the transformation algorithms. Since the characteristics of the emulated PDP11/60 are indistinguishable from the characteristics of the actual hardware, exactly the same code runs on both the actual hardware and the emulated machine.

This is not only a useful verification technique, but provides a means for verifying software interactions with hardware, such as device drivers whose performance is critical to efficient system execution.

3. RESULTS

The results of the LSI Applications project consist of:

1. Device design selection,
2. PDP11/60 emulation,
3. Eclipse C-300 emulation, and
4. Device simulation.

The first result is reflected in the type B-1 specification, while the remaining three are reflected in the SMITE software developed during the project and described in the LSI Applications Project Emulation User's Manual.

3.1 DEVICE DESIGN

The design approach selected as the result of the analyses of the algorithms and available architectures is the microprogrammed circuit architecture. The proposed microprogrammed circuit is called the Peripheral Transformation Processing Unit (PTPU).

The PTPU provides a degree of flexibility not possible with hardwired circuits, for example, approaching that generally found only in microprocessor-based systems. Probably the most evident advantage of this approach, and a primary factor in choosing it, is the ability to execute any algorithms on the same hardware elements merely by changing the controlling microcode.

The PTPU approach has several associated advantages. While another approach, such as the hardwired circuit, may provide a higher degree of performance, this approach easily meets and significantly exceeds the performance goals as was shown in paragraph 2.2.5.4.2.

An important advantage of the microprogrammed circuitry approach is the ability to share common resources, such as the basic hardware arithmetic functional units like the multiplier, adder/subtractor, and divider. This advantage is particularly important in view of the fact that the predominant processing operations involved in execution of the transformation algorithms are these basic arithmetic operations. Thus the optimization of these basic units becomes more noteworthy a task as the ability to increase their duty cycle increases.

Since processing control of the PTPU is effectively under the control of a software program, altering the functional configuration of the device, that is, altering the type of processing performed by the device, becomes a matter of altering the software that controls execution within the PTPU. Related to the feature of alteration of functional configuration is the fact that software control of the device has the added advantage of speeding all phases of development of the device, since the device may be developed and tested module by module by appropriate selection of controlling microcode.

This modular nature of the device has further benefits besides those realized during development and test. Modular architecture, as with modular software, has the feature that function is isolated to a particular section of the overall system. Thus identification of faults in the system reduces to an analysis of the symptoms and an isolation of the particular module whose function (or malfunction) is related to the pathological symptoms. Repairs may be easily effected by simply replacing the entire suspect module with a module known to be functioning properly. This feature reduces downtime from possibly days down to minutes, an important consideration from the point of view of system throughput.

3.1.1 Performance

Besides the advantages of this approach as discussed above, the degree of performance possible with this relatively simple architecture is particularly attractive. Table 2-5 showed the expected time for execution of the four major algorithms and their inverses. As a comparison, this table also shows the measured time for execution of these eight routines on the (emulated) PDP11/60. The expected execution time of these algorithms by the device, clearly well above the tenfold improvement goal, is obtained by the simple serial processing of the algorithm without any degree of concurrency involved. This important point shows that the DMA's goal may be obtained without the need to resort to highly complex parallel-processing architectures. Continuing this argument, the design which does utilize parallel processing would be expected to provide a significantly greater improvement in performance (at the cost of more complex circuitry).

However, since the desired performance level is more than amply met with this relatively simple "serial" processing, the proposed design focuses on a serial-processing architecture. While serial processing is sufficient to achieve the goals, the various considerations that must be observed are discussed in the appropriate sections that follow.

3.1.2 Host System-level Design

From both the applications and system analysts' point of view, the device should not affect the existing operational environment at the existing facility. This section will describe the nature of the device and its characteristics with respect to the anticipated operational environment.

The key to successfully imbedding the device within the existing host machine's operations is the selection of a mechanism for interfacing the device with the host. The most suitable selection is one that uses one of the various peripheral interface boards that are available from the host machine vendor. Besides providing a standard for the particular host machine, the vendor-supplied peripheral interface boards has an associated device handler containing the appropriate "hooks" for making the device recognizable to and compatible with the host operating system. Thus use of the DR11-B direct memory access peripheral interface board for the PDP11/60 and use of the 4041 interface board for the Eclipse C-300 has the dual advantage of:

1. Providing a documented standard for forming electrical and functional connection with the host bus, and

2. Providing an appropriate device handler.

For actual implementation of the device handler, a generalized form of the device handler, which is generally supplied by the software-systems vendor, would be patched to account for the particular device addresses associated with the device's controller board.

The use of a standard interface and device driver is particularly significant from the DMA's point of view. Use of a standard vendor-supplied

hardware peripheral interface board will enable continued maintenance of devices attached to the bus since the new device will be a vendor-recognized part. Similarly, no modifications of the operating system will be required, other than a possible vendor-supplied patch, to make the new device driver a recognized system routine, thus preserving the integrity of the operating system from the vendor's point of view. As a practical matter, it is anticipated that the change will be limited to one additional parameter that will be incorporated into the SYSGEN procedure.

Furthermore, since the new device driver is just another system device handler, using the device becomes straightforward from the software point of view. For example, consider the DR11-B DMA interface board for attaching the PTPU to the PDP11/60. Since this is the standard DMA controller used by the vendor for high speed devices, such as disks, the associated device driver, and thus the device itself, would be perceived as another high speed destination for and source of data. For the further purpose of explanation, assume a disk driver is modified to handle the PTPU. Then software access to the PTPU would be performed in the same way as file access to a disk is performed. A further implication is that the PTPU could be used at the system-command level. That is, by using the operating system's file-transfer utilities, a file of the appropriate format of data to be transformed could be transferred from a disk to the PTPU, then the transformed data could be transferred back from the PTPU and placed in a new file on the destination disk. Although this mode of operation might only be used for test purposes, the ability to function in this manner is an indication of the high degree of compatibility with existing software that this approach enjoys.

The normal mode of operation would be expected to be through applications software. The critical element in this path is a library of routines that form the link between high-level user-application software and the low-level device handler. Ensuring that this library contains the same entry points as the currently-used library of (software-only) transformation routines guarantees the full compatibility of the use of the PTPU with existing applications software.

Figure 3.1 describes the process flow involved in using such a library and device handler. This same process flow, or slight modifications of it, would apply to all the various types of transactions involving the PTPU.

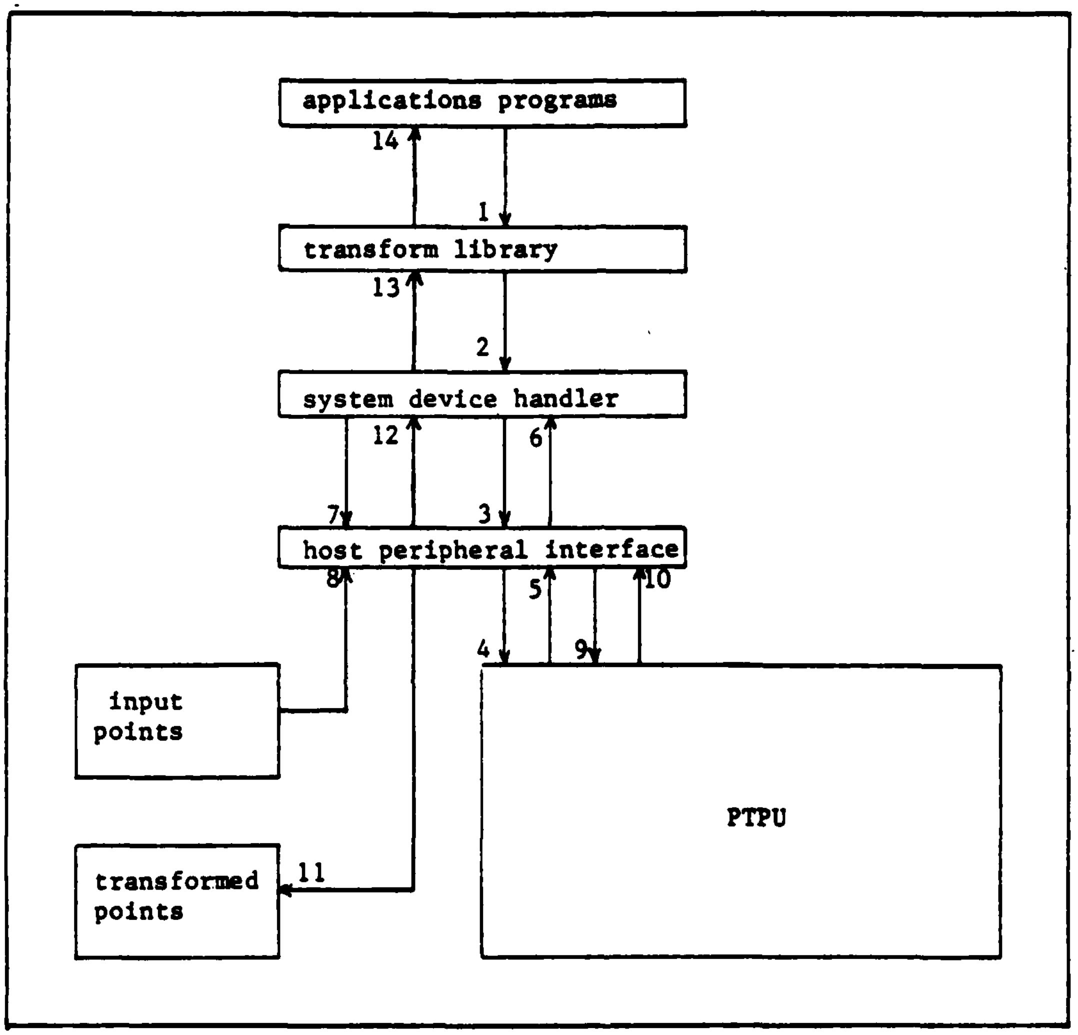
The microprogram control unit (MCU) controls internal processing in the PTPU. Through execution of microcoded instructions contained in storage within the MCU, the unit is responsible for process control required for the execution of the algorithms.

The arithmetic processing unit (APU), which is entirely host independent due to the effective buffering by the other units in the PTPU, is the center of arithmetic processing in the PTPU. As well as containing the basic arithmetic functional units, this unit contains an input/output buffer for storing input data, algorithm constants, intermediate results, and the final transformed points for transmittal back to the host.

The interface control unit (ICU) is the primary depository of host-dependent function within the PTPU. This unit handles the functional and electrical interface with the host peripheral interface board. To remove host dependency from the APU, the ICU formats data from the host into the format internal to the APU, then reformats transformed data for return to the host.

Since the MCU is the primary site of "intelligence" in the PTPU, there is some degree of host dependency in the MCU for those functions involving control of function of the ICU by the MCU. Removing those ICU-control functions from the MCU, and inserting them into the ICU, would eliminate host dependency in the MCU. While increasing the processing power of the ICU in this way might add to the cost of the ICU, a much higher savings would be realized in the overall system design, particularly when the PTPU is being considered for attachment to some other (unspecified) processor.

The modular approach avails ease of attachment to different hosts. By designing the MCU and APU as host-independent units, the PTPU then becomes a device with two major sections. The first major section, which includes the MCU and APU, is the same section as that on any particular



1-ap requests transform
2-tl issues request
3-handler sets command in interface
4-interface issues request to PTPU
5-PTPU indicates it's ready
6-interface notifies handler (interrupt)
7-handler sets DMA address and word count
8-DMA transfer passes through interface
9-data enters PTPU, transform begins
10-PTPU returns transformed data
11-DMA transfer begins
12-transfer done, handler notified (interrupt)
13-tl notified of finished transform
14-ap notified of finished transform

Figure 3.1. Systems-level Description of Host/PTPU Interaction

realization of the PTPU on any host. The other major section, the ICU, is the only section that is custom tailored to meet the requirements of the particular host. The only limitation to the effectively total generalization of this principle is the limitation of the precision carried by the APU. This issue is discussed further in Paragraph 3.1.2.2.

Figure 3.2 is an illustration of the process flow through the PTPU for a transaction involving a request for transformation coming down from the host. Figure 3.1 previously described the corresponding process flow within the host system.

Figure 3.3 describes the process flow for a transaction involving downloading of microcode from the host into microcontrol store in the MCU.

Figure 3.4 describes the process flow for a transaction involving a request for PTPU-internal status.

From the system level, downloading capability places the following additional requirements on the PTPU:

1. Read/write control store in the MCU, instead of or in addition to read-only control store;

2. Formatting capability within the ICU (besides that required for formatting transform data) to create the n-bit wide word (estimate 96 bits) for the microstore out of the 16-bit word of the host computer;

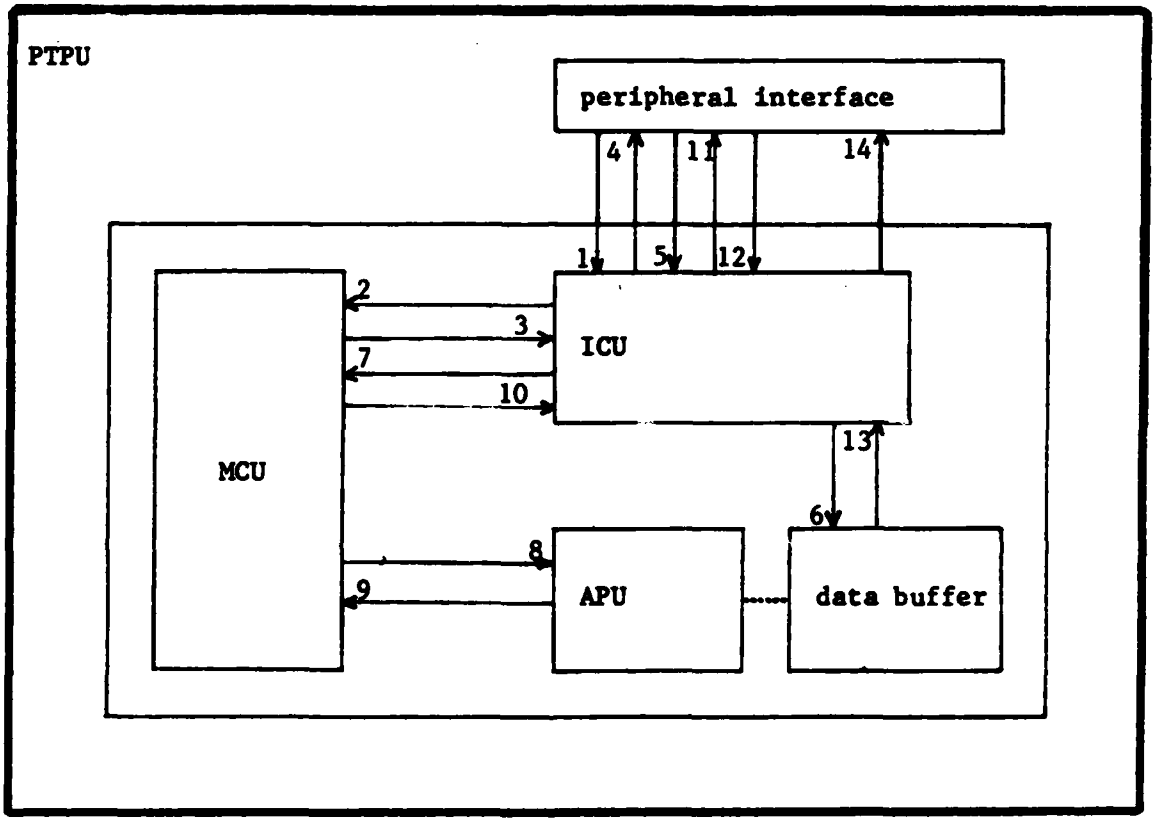
3. A data bus for transferring this control word from the ICU to the MCU, (note, however, that this data bus need not be n (96) bits wide);

4. A control subroutine in the MCU to execute the downloading, most probably contained in ROM; and

5. At least two status bits: one to indicate to the ICU that the incoming data is to be considered downloaded code, and a second to indicate to the host that the transfer has terminated successfully.

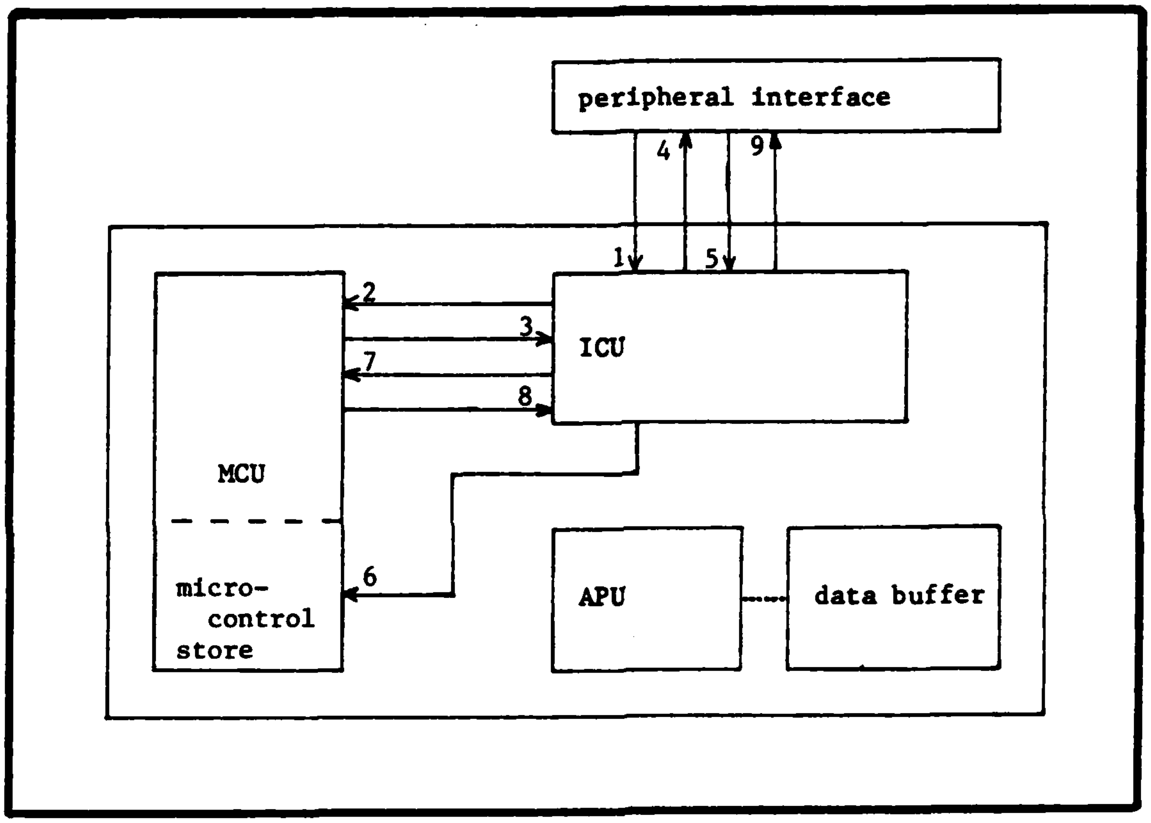
These additional requirements do not significantly impact the basic design since:

1. The type of control store (RAM or ROM) is transparent to the design.



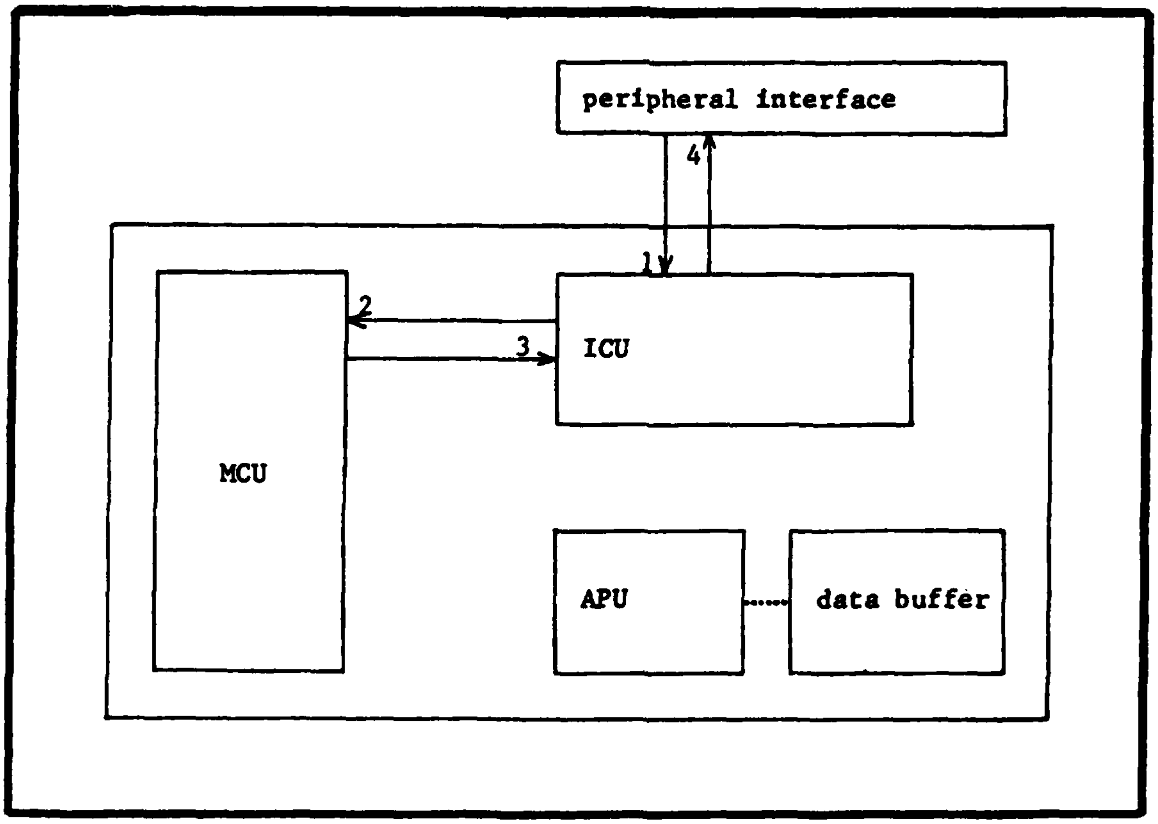
1-request to send data
2-request for incoming data
3-ok to accept
4-ok to send
5-data transfer and formatting
6-formatted data passed to buffer
7-transfer complete
8-execute transformation
9-transformation complete
10-send transformed data
11-request to send
12-ok to send
13-get transformed data for reformatting
14-send reformatted data

Figure 3.2. PTPU Process Flow for Transforming Data



1-request to send microcode
2-request for incoming microcode
3-ok to accept
4-ok to send
5-microcode accepted and formatted for MCU
6-formatted microcode loaded in MCU microstore
7-transfer complete
8-transfer successful
9-transfer successful

Figure 3.3. PTPU Process Flow for Downline Loading Code



1-request for status
2-request for status
3-status information
4-status information formatted and transmitted

Figure 3.4. PTPU Process Flow for Status Request From Host

2. The ICU is already required to contain formatting capability;

3. Control lines already exist between the ICU and MCU; n additional lines represent 2n buffers, which is not a significant hardware load to add (The actual width, n, of the extra bus becomes a matter of determining the relative importance of speed of downloading vs. cost of implementing the bus.);

4. Control subroutines already exist in the MCU (The requirement of placing the download-control routine in ROM was discussed in Item 1.); and

5. Status bits already exist in the PTPU (Two extra bits are not a significant addition.).

The additional capability afforded the device by the download feature far outweigh these additional requirements on the design.

If download capability is rejected in favor of ROM control store, the control store must be of sufficient size to permit several algorithms to be stored simultaneously. Ideally, the four major algorithms and their inverse would need to be maintained simultaneously. Failing to store the commonly-used algorithms simultaneously would guarantee increased downtime since the device would have to be shut down to physically replace the microstore ROM with the ROM containing the desired algorithm. On the other hand, downloading into RAM would permit a reduction in the required size of the control store, since the desired routine could be loaded as needed.

3.1.2.1 Microprogram Control Unit (MCU)

The MCU is the "CPU" of the PTPU in the sense that this unit is the source of control of processes in the PTPU. By responding to status signals received from the other units in the PTPU, the MCU coordinates their activities for the reliable execution of routines stored in the MCU control store memory area. Other than host dependencies relating to control of the ICU, the MCU is a host-independent unit. Thus, neglecting for the moment the ICU-related issues, the MCU does not require any reconfiguration to attach the PTPU to a different host.

The architecture of the MCU is optimized for speeding control-signal generation and also for minimizing control store. The function of the unit

is to provide control signals to the ICU and to the APU, in this way controlling data transfer and processing activities of the device. Figure 3.5 shows the hardware details of the MCU.

The major components of the MCU are:

1. Clock circuit,
2. Micro control memory (control store),
3. Address selection logic circuit, and
4. Control signal register

3.1.2.1.1 Clock Circuit. The clock circuit will generate the four clock signals used by the four different microoperations of the MCU.

3.1.2.1.2 Micro Control Memory (Control Store). The microcontrol memory stores the control programs required for operation of the PTPU as well as the microprograms corresponding to the transformation routines of the Defense Mapping Agency. The size of the microcontrol store is estimated to be 96 bits wide and approximately 4000 locations deep. This memory will be constructed from high speed read-only memory and read/write memory in the case of implementation of downloading capability in the PTPU.

3.1.2.1.3 Address Selection Logic. The address selection logic generates the next address for the microcontrol memory.

3.1.2.1.4 Control Signal Register. The control signal register is loaded from the microcontrol memory and generates all the necessary control signals corresponding to the microprogram stored in the microcontrol memory. The control signal register might be partitioned into the bit fields defined in Table 3.1.

3.1.2.2 Arithmetic Processing Unit (APU)

As the central repository of all mathematical processing in the PTPU, the APU resembles an FPU in a computer system. However, not capable of any independent processing, the APU executes mathematical operations under control of the MCU. Because of the logical buffering by the MCU and ICU, the APU is host independent. The only restriction to this independence is the precision of the components of the APU.

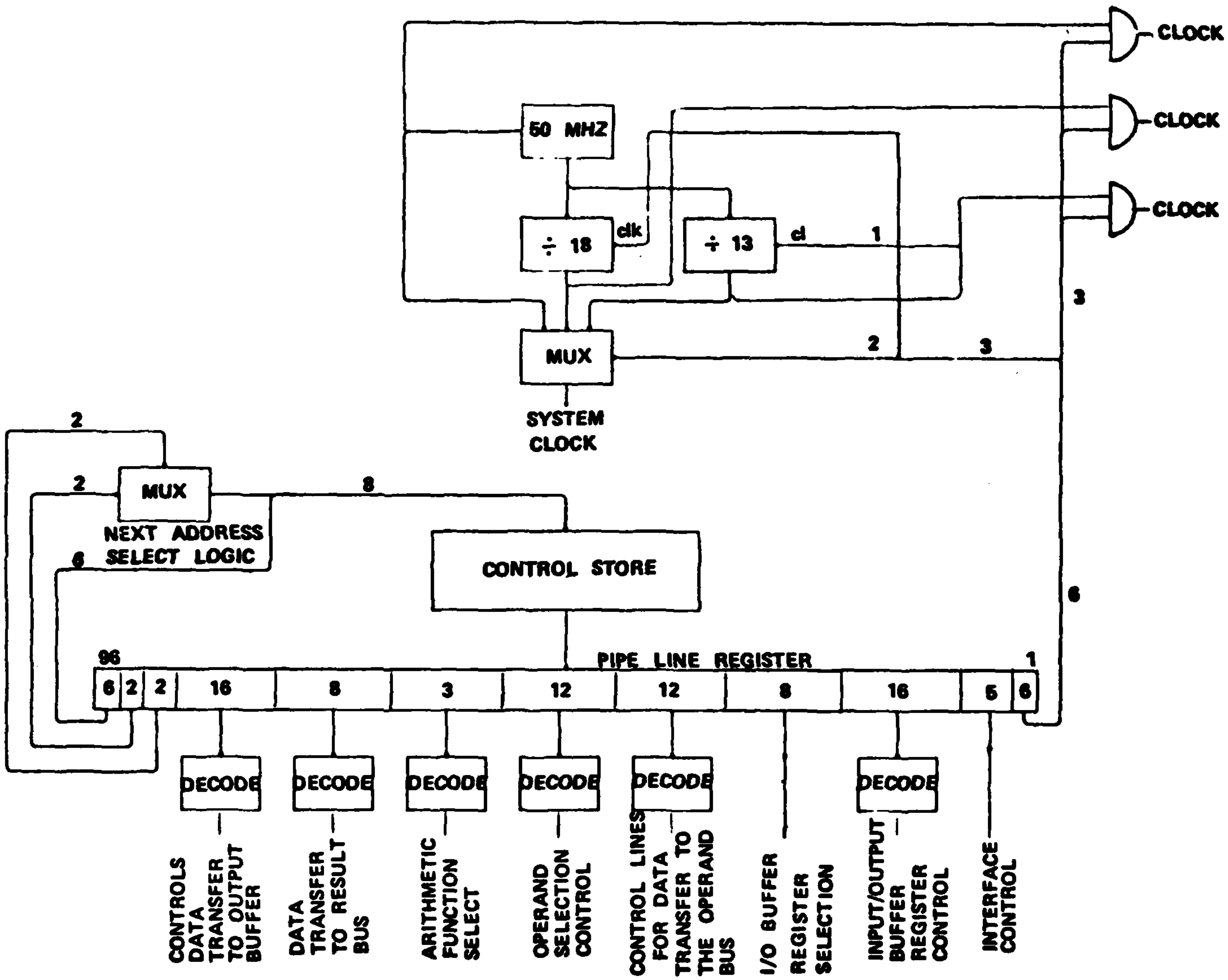


Figure 3.5. PTPU Microprogram Control Unit

Table 3.1. Microcode Definitions

bit	number of control lines	control signal definition
1-6	6	clock select
7-11	5	interface control
12-27	16	input/output buffer control
28-35	8	input/output buffer clock control
36-47	12	control lines for data transfer to the operand bus
48-59	12	operand selection control
60-62	3	arithmetic function select
63-70	8	data transfer to the result bus
71-86	16	controls data transfer to the buffer from the result bus
87-88	2	selects two least significant bits for the next address for the conditional jump
89-90	2	provides two least significant bits for next-address selection
91-96	6	provides six most significant bits for the next-address selection logic

Throughout the project, the assumption has been that the PTPU would be attached to 16-bit machines with single-precision floating-point formats with no more than 8-bit exponents and 24-bit mantissas. Consequently, a single-precision floating-point format has been considered the internal format of the PTPU and discussion throughout this report will tend to reflect this consideration.

Using the PTPU with equal to or less than 32-bit single-precision floating-point accuracy will not impact design of the PTPU. The ICU was designed to handle this contingency.

In the case of greater than 32-bit single-precision floating-point values, the design of the PTPU is impacted by two additional requirements:

1. Increased precision in the basic mathematic operation hardware units, and

2. Increased width of data buffers in the APU and associated busses within the PTPU.

Using the double-precision feature of the PTPU is one method of achieving higher precision for extended precision hosts; however, this approach could adversely affect performance since the PTPU design has been optimized for 32-bit formats.

The architecture of the APU is optimized with respect to the characteristics of the DMA transformation routines as determined by analyses described previously. The APU performs all arithmetic functions for the PTPU and consists of the following major components.

1. Input/output buffer,

2. Input operand bus,

3. Input holding registers,

4. Arithmetic functional units,

5. Result holding register,

6. Output operand bus.

Figure 3.6 illustrates the hardware details of the APU.

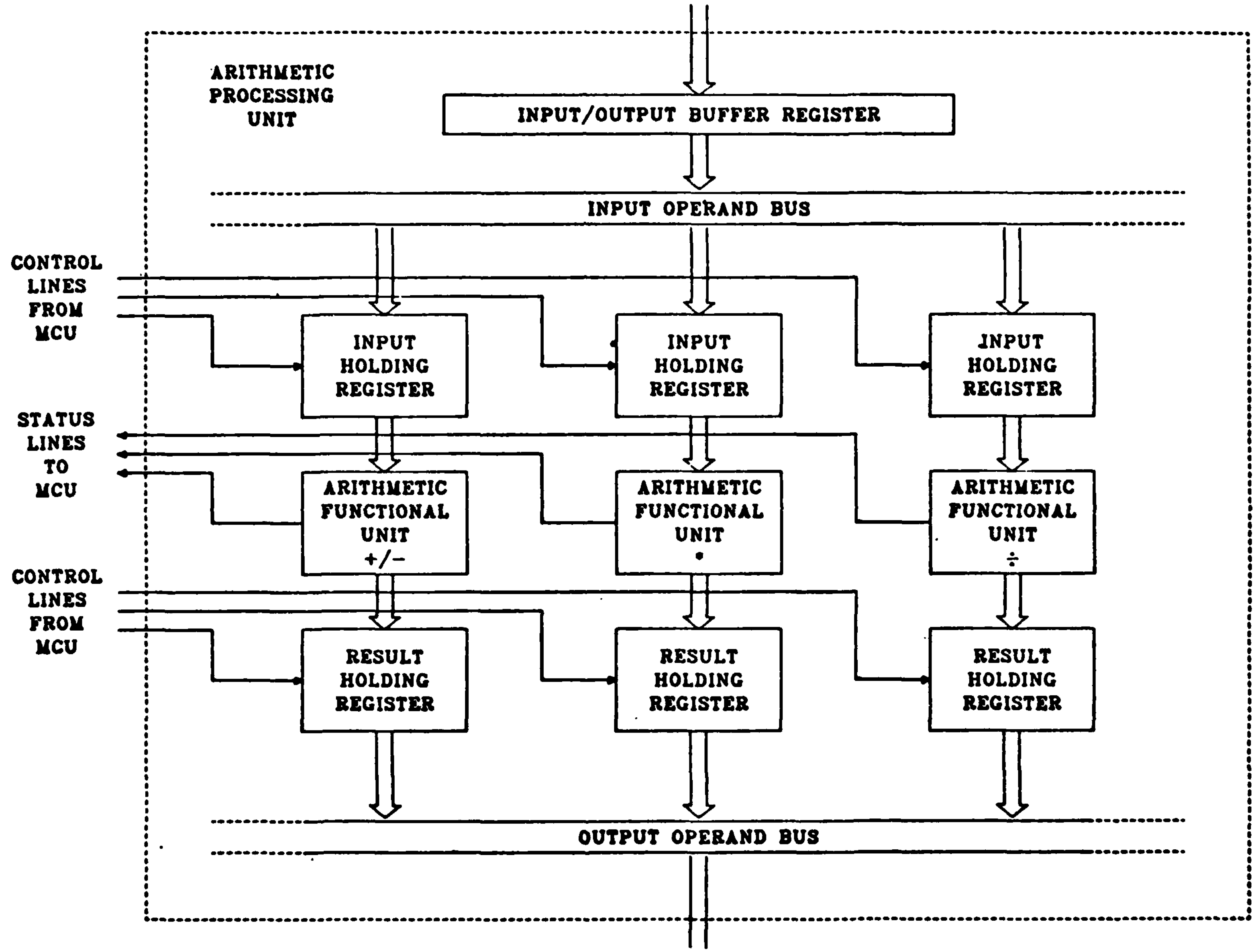


Figure 3.6. PTPU Arithmetic Processing Unit

3.1.2.2.1 Input/output Buffer. The input/output buffer receives data from the host computer via the ICU. This buffer is the main data storage area of the device. Each storage area, or word, in the buffer will be of the form shown in Figure 3.7.

The total number of words in this buffer is a function of the expected data throughput rate, the expected data transfer block size, and the degree of contention that can be tolerated on the host bus. At low throughput rates, transferring individual words at a time will not significantly increase bus contention. As throughput rate increases, the burst mode of DMA transfer will need to be utilized to decrease bus access protocol overhead, and, therefore, bus contention.

A block size of 4000 words (1000 points) would be a reasonable upper limit for a transfer block size. Larger block sizes could cause the bus to be unavailable for unacceptably long periods of time, since burst DMA does not release the bus until the entire transfer is completed. Assuming a 4K word working block, the APU data buffer should be 4·4K words in length. This allows for the necessary separation of input and output buffers, and provides double buffering for maximum parallel operation of the separate units of the PTPU. Figure 3.8 schematically illustrates the logical partitioning of function among the four 4K word buffers in the APU.

3.1.2.2.2 Input Operand Bus. The input data from the buffer are transferred to the input holding registers via the input operand bus under control of the MCU.

3.1.2.2.3 Input Holding Registers. The holding registers act as buffers for the operands used by the arithmetic functional units. Each input to an arithmetic functional unit will be driven by an input-holding register. Their inclusion allows the possibility of parallel use of different arithmetic functional units by the MCU.

3.1.2.2.4 Arithmetic Functional Units. The arithmetic functional units provide all the arithmetic operations used by the device.

1-bit sign	8-bit exponent	24-bit mantissa

Figure 3.7. APU-word Format

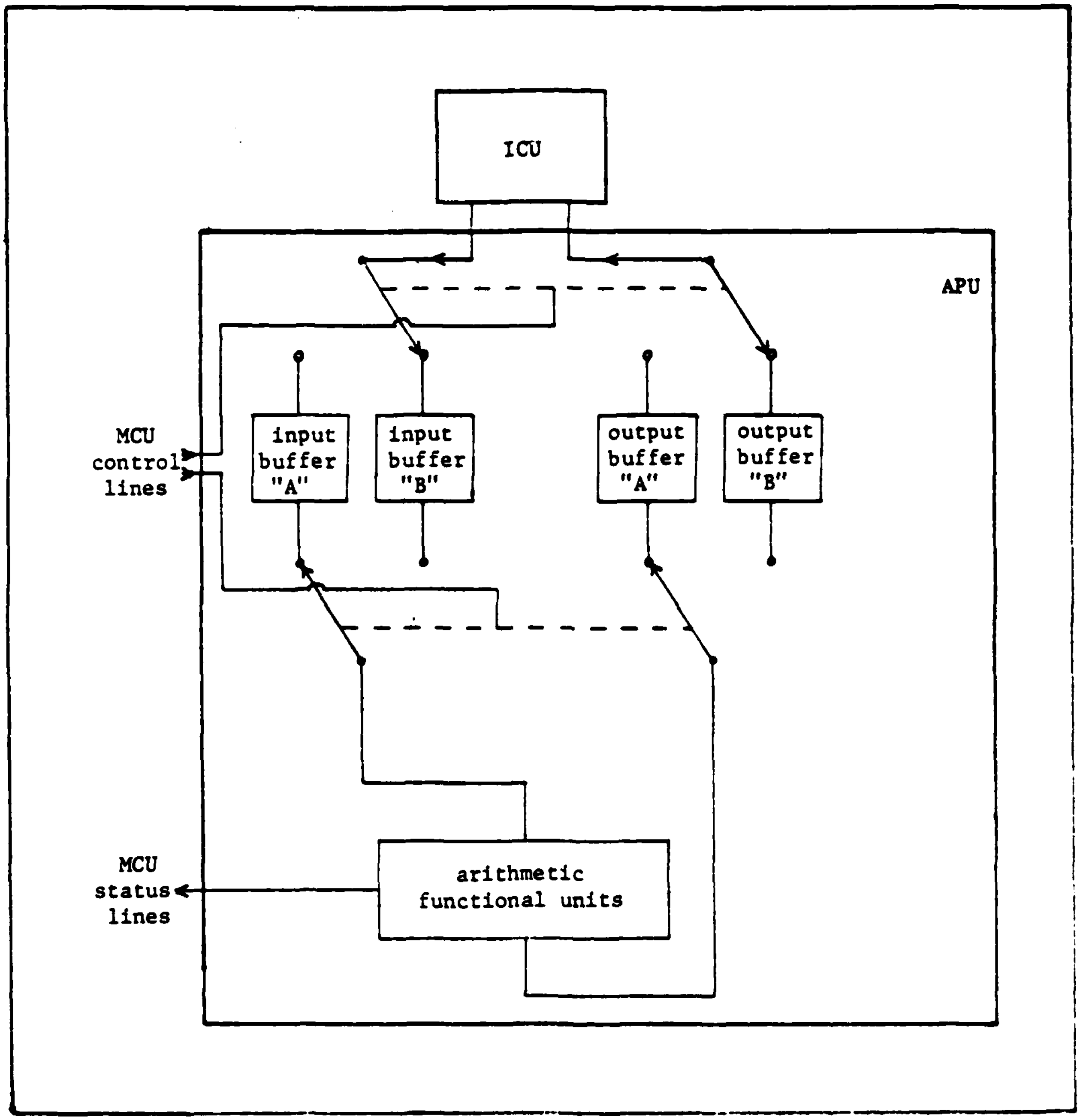


Figure 3.8. Double-buffered I/O in the APU

3.1.2.2.4.1 Types of Arithmetic Functional Units. The arithmetic functional units will be of three types:

1. Addition/subtraction,
2. Multiplication, and
3. Division.

These units correspond the "simple" operations discussed previously. "Non-simple" arithmetic operations, such as trigonometric functions, will be performed by microprogram subroutines using the arithmetic functional units. The subroutines will be polynomial expansions of the desired function, carried to sufficient accuracy to match the precision of the device.

3.1.2.2.4.2 Performance of Arithmetic Functional Units. Using technologies already developed at TRW, the functional units will provide the execution speeds listed in Table 2.3. All of the functional units will operate on data of the form shown in Figure 3.7.

3.1.2.2.4.3 Number of Arithmetic Functional Units. The number of units will be one each for each of the three possible types. As already demonstrated in Table 2.5, this restriction to serial processing of the algorithms is sufficient to meet the order of magnitude performance goal. The use of the holding register before and after each arithmetic functional unit provides a mechanism for incorporating several units of each type for concurrent execution of the basic mathematical operations, should a determination be made later that the extra performance afforded by concurrent processing is desirable.

3.1.2.2.5 Result Holding Register. The output from an arithmetic functional unit will be buffered by a result-holding register.

3.1.2.2.6 Output Operand Bus. Transformed data are transferred from the output-holding register to the I/O buffer via the output operand bus under control of the MCU.

3.1.2.3 Interface Control Unit (ICU)

The ICU corresponds to the peripheral processor used in many computer systems to isolate transactions with peripheral devices to a single unit. As such, the ICU is the major site of host-dependent hardware.

Since the ICU supports electrical and logical connections between the PTPU and the host computer, its configuration is highly dependent on the host peripheral interface board with which it communicates.

Within the ICU's control is the formatting of data for providing compatibility between the data format used by the host computer and that used by the arithmetic functional units. The major components of the ICU (Figure 3.9) are:

1. A communication unit, and

2. A data-formatting unit.

3.1.2.3.1 ICU Communication Unit. This unit provides handshaking, logic, and electrical connections with the host computer peripheral interface board. Consequently, this unit passes status information between the host interface and the PTPU's MCU.

3.1.2.3.2 ICU Data-formatting Unit. Since the floating-point format for the two specified host computers is different, and no doubt will be different for future hosts, this unit provides a means of standardizing the data presented to the APU. The data-formatting unit will be a simple set of buffer registers with the appropriate bit-to-bit mapping required to move the required bits to the correct fields (Figure 3.10).

The follow-on designer may find it appropriate to include a set of high speed input and output buffers within the ICU to support this data-formatting activity. Due to the low cost of memory, its inclusion would have a negligible impact on device cost.

3.1.2.3.3 PTPU Attachment to Different Host Computers. As discussed in 3.1.3, the modular architecture of the PTPU provides a number of advantages in attaching the PTPU to a different host computer. By collapsing all host-dependent function into the ICU, the PTPU becomes a host-independent transformation processor (composed of the MCU and APU) that may be attached as needed through a particular ICU to whatever host requires the augmentation.

Thus, by maintaining an ICU for each applicable host, the same PTPU may serve a number of hosts simply by reattaching the particular ICU. If

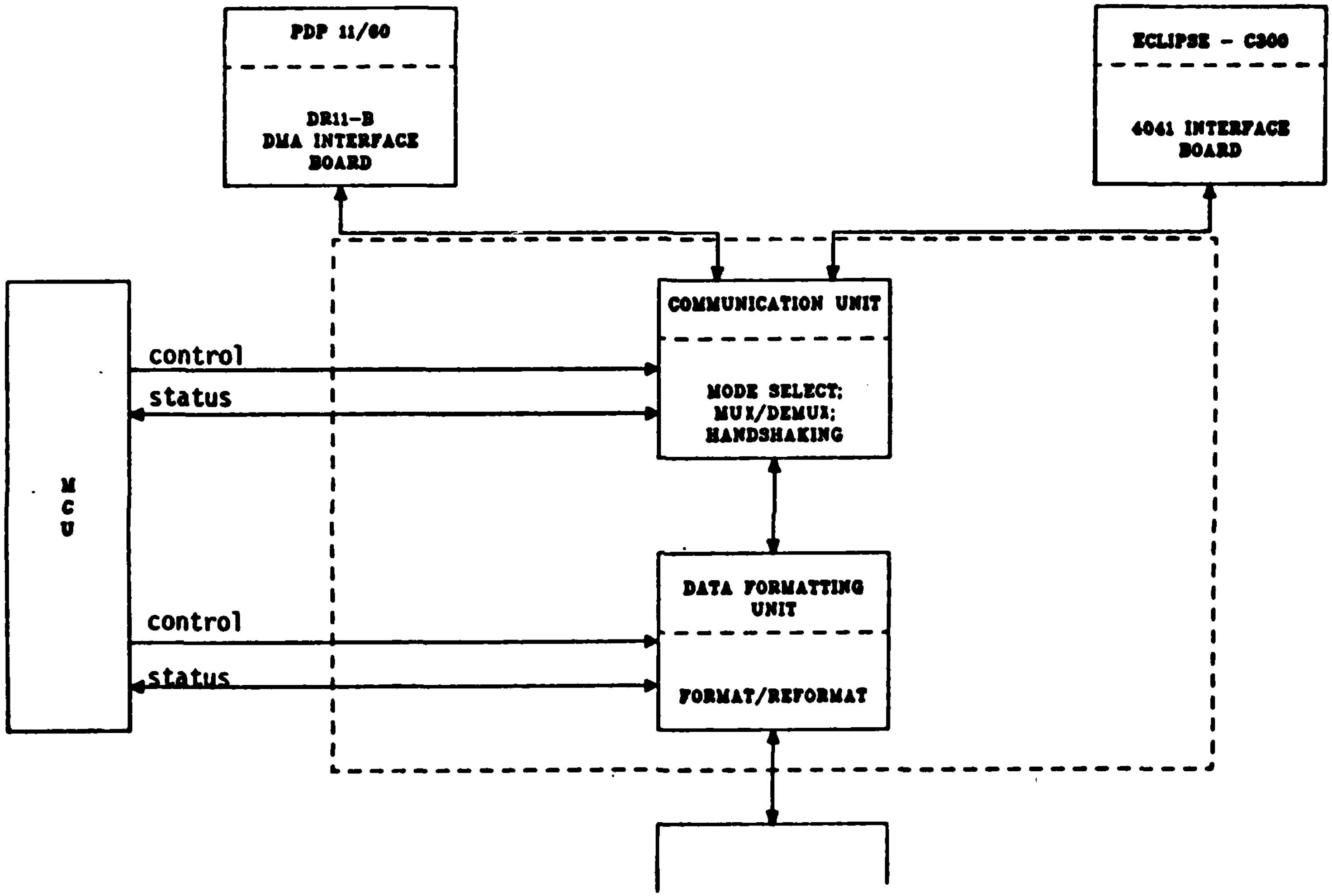


Figure 3.9. PTPU Interface Control Unit

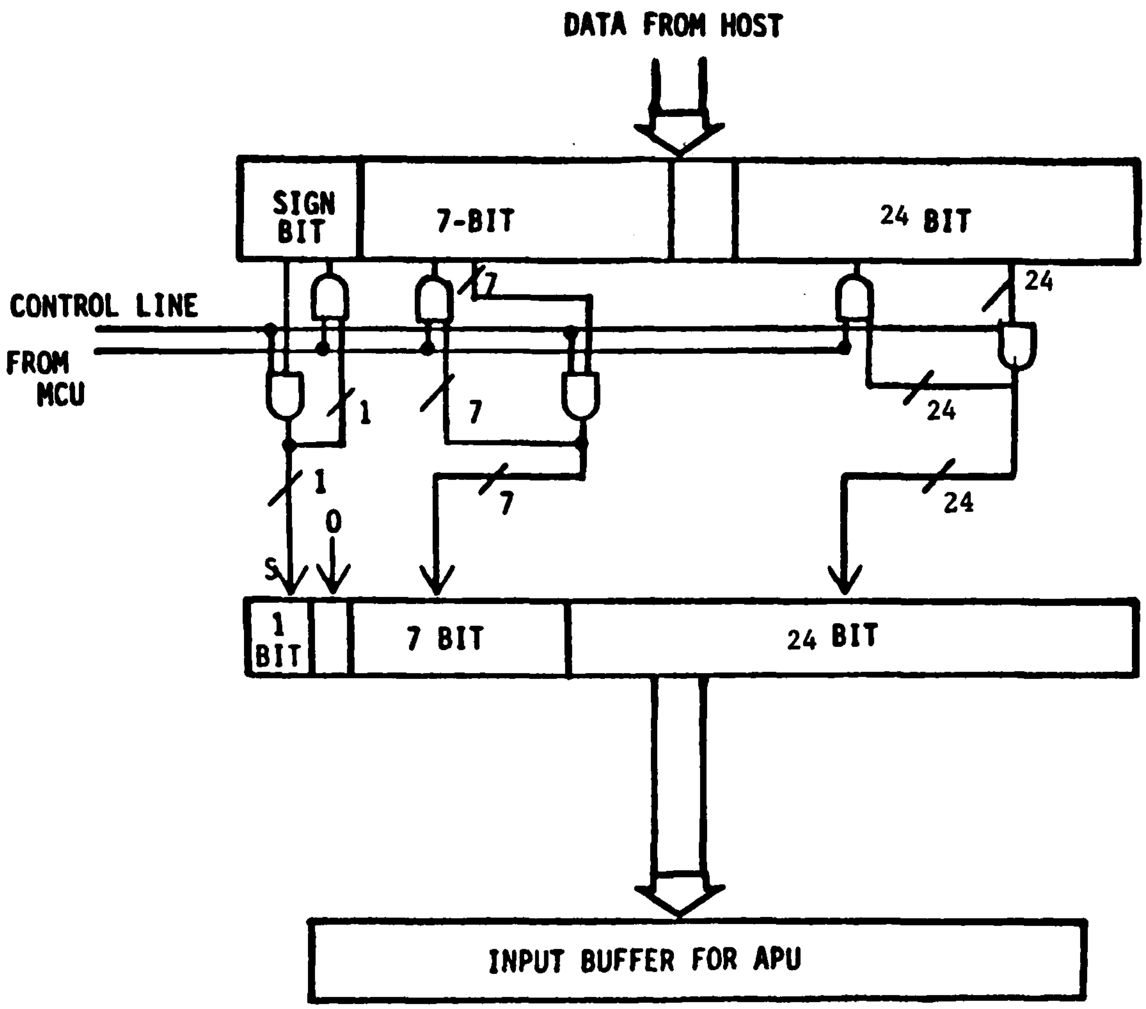


Figure 3.10. Data-Formatting Circuit

the device were later attached either to a different machine or to a different host peripheral interface board, the following modifications would be required at a minimum:

1. Replace ICU communication unit to match electrical and functional connections, and

2. Replace ICU data-formatting unit to provide the data format expected by the APU.

As discussed in 3.1.2.2, the design of the PTPU has focused on a format which has no greater precision than that provided by an 8-bit exponent and a 24-bit mantissa. If the PTPU were implemented with this internal format, the modifications described above would allow the PTPU to be attached to machines of the same or less precision. However, the modifications would become more profound as the PTPU was attached to machines of higher and higher precision. These modifications would eventually require modification of the basic arithmetic functional units themselves.

Due to the relative expense of 64-bit hardware arithmetic functional units, it may be more cost-effective to limit the "standard" APU to the 8-bit exponent and 24-bit mantissa described previously. A higher-precision APU could be designed for those machines requiring extended precision. The differences between the two versions of the APU would be higher precision arithmetic functional units, wider memory, bus, and associated buffer/driver paths. This dual-APU approach is viable only because of the modular nature of the PTPU architecture.

3.2 EMULATION

Concurrently with the development of the PTPU device specification, SMITE descriptions of the host computers were created for executing emulations of the host computers on the Nanodata QM-1. Development of the SMITE descriptions involved analysis of the instruction sets and associated operation codes. Using vendor-supplied execution times for the various micro-instructions, the emulations faithfully observed the relative timing of the various instructions. In fact, their use provided the performance measurement of the transformation algorithms used as the baseline performance level in this project.

Since SMITE has been a developing research tool, it's use during the course of this project was its first large-scale application, and unsurprisingly it uncovered a number of small problems which either delayed the progress of the development of the emulations, or required development of methods to work around the problem.

One of the more vexing problems encountered throughout the project was the frequent errors encountered in transferring compiled emulations to the QM-1. Incompatible with ease of use as a development tool, the QM-1 interface for file transfer often had to be repeated a number of times (at a cost in time of approximately one half hour per transfer) before a successful transfer was accomplished.

Since SMITE is only now emerging from the R&D stage, its use in major emulations such as the PDP11/60 and Eclipse C-300 generated problems associated with the sheer size of the descriptions. Since list space in the compiler proved to be too small, some of the descriptive qualities of SMITE were lost in working around the problem. For example, since there was insufficient space to name the individual elements of the description, many data items were placed in one array, which required only a single name.

Similarly, the size of the emulations exceeded the memory capacity of the QM-1. Methods to work around this problem until more memory was attached were successful, but at the cost of the software structure of the emulation.

3.2.1 PDP11/60 Emulation

The SMITE description of the PDP11/60 contains the full instruction set complement and the floating-point unit. This emulation, running on the QM-1, was used to measure execution times of the four major transformation algorithms and their inverses as listed in Table 2.1. Because of the 20% error in vendor-supplied values for timing of the various operations on the PDP11/60, the values listed for these algorithm execution times are accurate to only within 20%.

As previously described, these four algorithms account for approximately 90 percent of the processing load. Furthermore, after a detailed and

in-depth review of the other remaining algorithms, it was determined that all of the constructs, functions, and operations were demonstrated by the four selected algorithms. Therefore, it became unnecessary to repetitively measure the execution times of all of the algorithms.

In addition, a Unibus description was included to support simulation of the bus traffic generated by DMA controller for the PTPU. Since the emulation runs under SASS, all the appropriate performance measurement hooks are available.

The details of the emulation are found in the SMITE descriptions and in the Emulations User Manual.

3.2.2 Eclipse C-300 Emulation

The SMITE description of the Eclipse C-300 contains the full instruction set complement and the floating-point unit. Furthermore, since the PDP 11/60 execution time clearly demonstrates the necessary improvement and since the PDP 11/60 is the more powerful processor (for the types and numbers of operations to be performed in the DMA transformation algorithms), it is clear that the speed-up will be even more pronounced on the Eclipse C-300. Therefore, it became unnecessarily repetitive to perform a detailed emulation of the algorithms to demonstrate satisfaction of the requirements with the C-300 emulation. While the four major transformation algorithms and their inverses were not executed on this emulation, the correct operation of the individual instructions and the floating-point unit were validated. Since the emulation runs under SASS, all the appropriate performance measurement hooks are available. The details of the emulation are found in the SMITE description and in the Emulation User Manual.

3.2.3 Device Simulation

Since the PTPU is a microprogrammed unit, a true emulation of the device would require emulation of the PTPU down past detail of the MCU down to the bit level of the control-signal register. Instead of pursuing this detail, a host-bus level description was formulated for the PDP11/60 and DR11-B controller (Figure 3.11).

Due to space restrictions, the FPU was not emulated simultaneously with the PTPU controller. Figure 3.12 describes the detail of the cycle-by-cycle description of an interaction between the host and the controller.

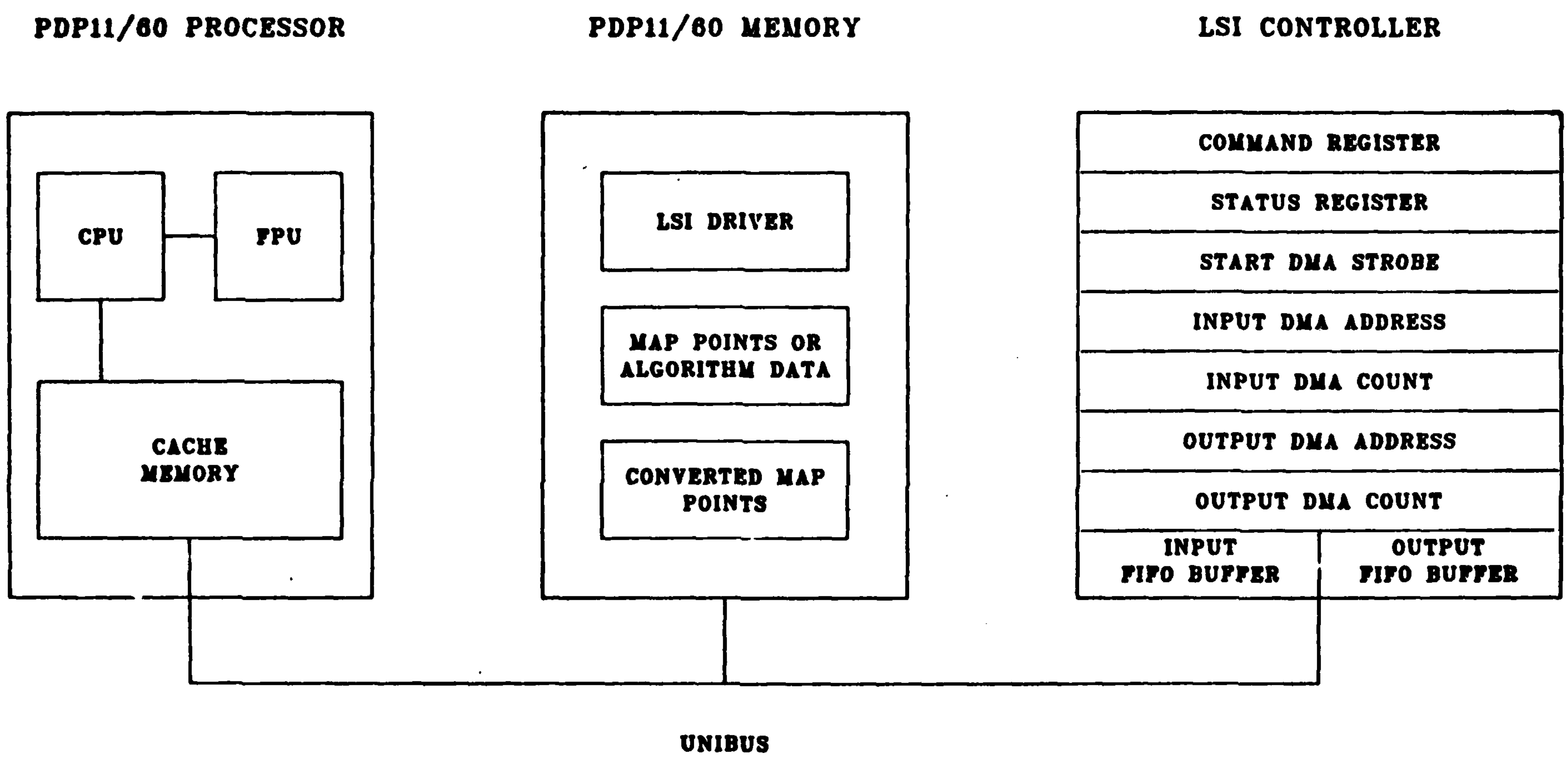


Figure 3.11. PDP 11/60 PTPU Controller Emulation System Overview

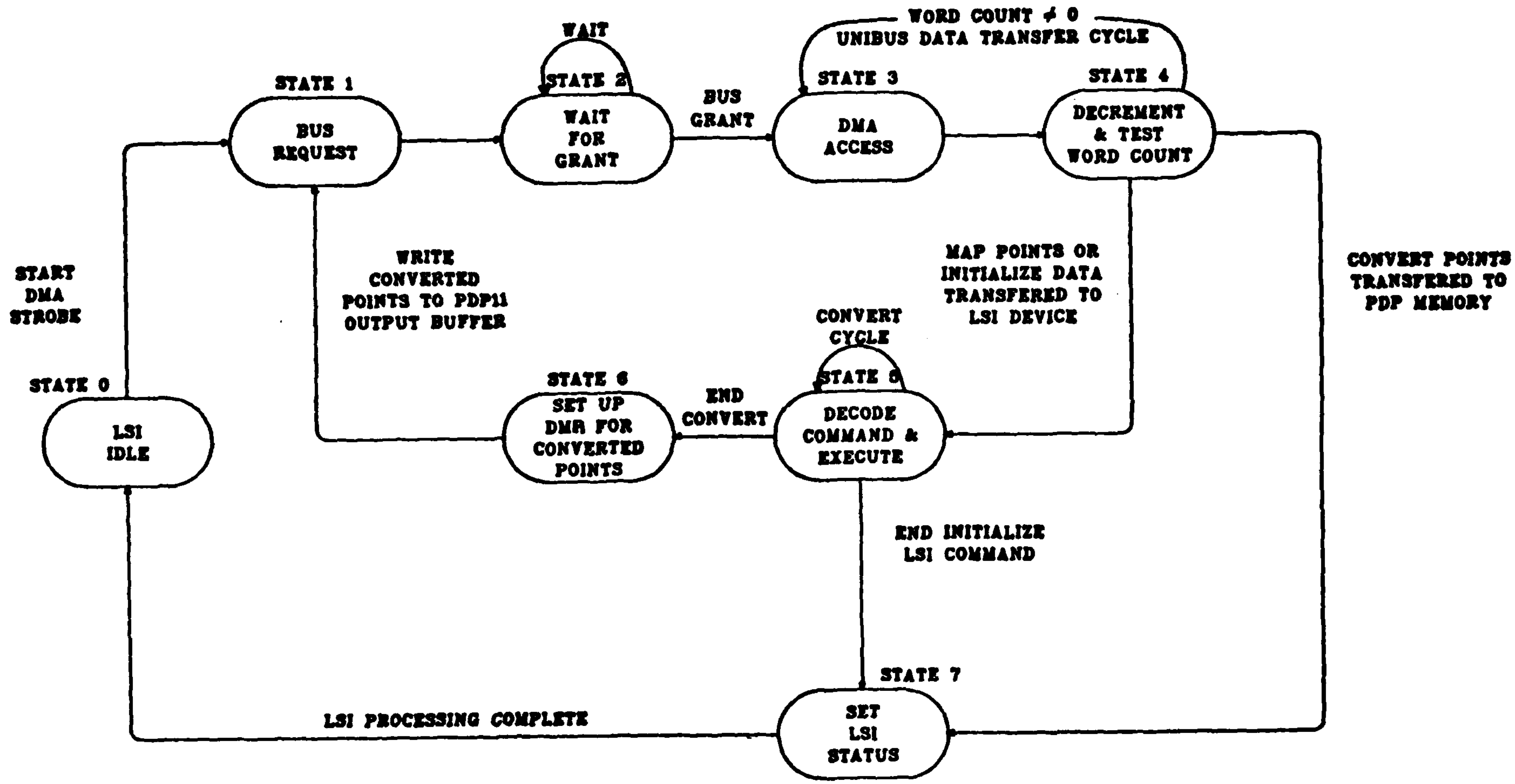


Figure 3.12. PDP 11/60 PTPU Controller Emulation Bus Interaction Design

As noted previously, since the emulation runs under SASS, all the appropriate performance-measurement hooks are available. Details of the controller emulation are contained in the SMITE description and in the Emulation User Manual.

4. CONCLUSIONS AND RECOMMENDATIONS

A number of conclusions regarding the optimal design of the PTPU to support the stated performance goals were brought out in the course of this report. This section will state the system-level conclusions and recommendations regarding the proposed design and follow-on efforts.

4.1 DEVICE PERFORMANCE

As demonstrated in this report, the proposed microprogrammed circuit, called the Peripheral Transformation Processing Unit (PTPU), will execute the transformation algorithms at a rate several times faster than the stated performance goal of an order of magnitude improvement over that of the host computers. While other architectures may provide equal or even faster execution of the algorithms, the proposed architecture meets the highest level of flexibility, maintainability, and optimal allocation of resources. Furthermore, the proposed architecture is a proven approach, eliminating a prolonged or high risk design cycle.

4.2 EMULATION

Emulation has been shown to be an extremely powerful aid in the design and development of hardware designs. Its use, by means of the hardware description language SMITE, should be used early in the design and development phase of hardware definition. Early development of a SMITE description would provide:

1. Architecture documentation,
2. Identification of ambiguities or holes in the architecture definition, and
3. A mechanism for conducting performance tradeoffs early in the design phase when the tradeoffs could be easily incorporated into the design.

As the hardware definition stabilizes the development of a prototype device commences, software developers may use the emulation to gain experience with the architecture, to perform algorithm analysis and tradeoff studies, and to initiate low-level testing before the hardware is actually available.

4.3 FUTURE CONSIDERATIONS

During the course of the project, several items clearly stood out as helpful to the development effort, but due to the limitations of the present project, had to be set aside for future consideration. None of these items affected the design of the proposed PTPU device; however, the final design might have been realized more expeditiously had these items been available.

4.3.1 SMITE

A number of problems in the current implementation of SMITE were discussed in 3.2. Besides providing solutions to the stated problems, there are a number of system-level issues whose treatment will prove beneficial to the type of problem handled in this project.

Development of a hardware definition with SMITE should proceed prior to the hardware specification phase. The limited period of time available in this project required the concurrent development of the host computer emulations and the hardware specification. Consequently, the emulation occupied a backup position, rather than a front-line position as it is clearly capable of doing.

Emulating a microprogrammed architecture is a very large task. However, the eventual execution of this task will prove extremely beneficial to the hardware development effort.

The SMITE compiler should be available on machines providing easier access for the SMITE description developer. The logistic problems discussed in 3.2 are not compatible with the desired features of a design and development tool.

4.3.2 Automated Firmware Design

Although the stated performance goals were met with a relatively simple architecture, the use of AFD should become a routine step in design of microprogrammed circuits. Since AFD is currently mainly a research tool, application of its processes to the present project would have entailed a relatively large effort, particularly in view of the fact that the selection of the number of arithmetic functional units was not a difficult choice in this instance.

In fact, TRW possesses a unique set of software aids, each of which is individually powerful enough to carry the burden of a design effort on its own. However, the combination of AFD for design description, SMITE for design verification, and VLSI-design principles for device implementation will present TRW designers with extremely versatile and powerful mechanisms for rapidly converting arbitrary algorithms into finished hardware.

5. REFERENCES

5.1 GOVERNMENT DOCUMENTS

a. Rome Air Development Center, CP 078779600E, RADC Computer Software Development Specification, May 30, 1979.

b. Department of Defense, 7935.1-S, Automated Data Systems Documentation Standards, September 13, 1977.

c. Statement of Work for LSI Applications: PR No. B-9-3155, Rome Air Development Center, Griffiss Air Force Base, New York.

5.2 NON-GOVERNMENT DOCUMENTS

a. TRW, Advanced SMITE Training Manual, 32584-6015-RU-00, September 1, 1979.

b. TRW, PDP11/60 Emulation Requirements Specification, 35735-6003-UT-00, July 22, 1980.

c. TRW, LSI Users Manual, 35735-6015-UT-00, 27 May 1982.

d. TRW, Advanced SMITE SASS Software Requirements Specification, 32584-6004-RU-00, June 1978.

e. TRW, Advanced SMITE Compiler/SASS Interface Control Document, 32584-6005-RU-00, June 1979.

f. TRW, Advanced SMITE Compiler Maintenance Manual.

g. TRW, Proposal for the LSI Application Proposal 35735.000R1.

h. TRW, Development Specification of Peripheral Transformation Processing Unit (PTPU) for the DMA Transformation Routines (B-1 Specification) 25 March 1982.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED